Universidade de Aveiro

2009

Departamento de Electrónica, Telecomunicações e Informática

MICHAEL PHILIP SOUSA SHER

UNIDADE DE GESTÃO DINÂMICA DE TOPOLOGIA DE UM BARRAMENTO DE CAMPO

DYNAMIC TOPOLOGY MANAGEMENT UNIT FOR A FIELDBUS

Universidade de Aveiro Departamento de Electrónica, Telecomunicações e Informática

2009

MICHAEL PHILIP SOUSA SHER

UNIDADE DE GESTÃO DINÂMICA DE TOPOLOGIA DE UM BARRAMENTO DE CAMPO

DYNAMIC TOPOLOGY MANAGEMENT UNIT FOR A FIELDBUS

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Electrónica e Telecomunicações, realizada sob a orientação científica do Doutor José Alberto Gouveia Fonseca, Professor Associado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro e do Doutor Arnaldo Silva Rodrigues de Oliveira, Professor Auxiliar Convidado do Departamento de Electrónica, Telecomunicações e Informática da Universidade de Aveiro.

Dedico este trabalho à Alexandra e à minha mãe pelo apoio e motivação.

O júri / The jury

Presidente / President	Prof. Doutor Alexandre Manuel Moutela Nunes Mota Professor Associado da Universidade de Aveiro
Vogais / Examiners committee	Prof. Doutor Mário João Barata Calha Professor Auxiliar do Departamento de Informática da Faculdade de Ciências da Universidade de Lisboa
	Prof. Doutor José Alberto Gouveia Fonseca Professor Associado da Universidade de Aveiro (Orientador)
	Prof. Doutor Arnaldo Oliveira Professor Auxiliar Convidado da Universidade de Aveiro (Co-Orientador)

Agradecimentos

Ao meu orientador, Professor José Alberto Gouveia Fonseca, da Universidade de Aveiro, que contribuiu para o crescente gosto pela área da investigação e desenvolvimento.

Ao Professor Arnaldo Oliveira, da Universidade de Aveiro, pela colaboração e disponibilidade.

palavras-chave

Redundância e topologia em barramentos de campo, gestão dinâmica do barramento CAN, replicação e consistência da informação.

resumo

A redundância, mudança de topologia, detecção e isolamento de falhas pode aumentar a fiabilidade de um barramento de campo, como o *Controller Area Network* – CAN. A *Dynamic Topology Management (DTM)* em CAN faz uso da redundância e da detecção e contenção de falhas. Esta solução possui duas entidades fundamentais: a Unidade de Gestão de Topologia (*Topology Management Unit* – TMU) e a Unidade de Comutação de Rede (*Network Switch Unit* – NSU). A TMU é responsável pela gestão dos meios de comunicação e pelo acesso de cada NSU a pelo menos um meio de comunicação. A própria TMU possui redundância, havendo sempre uma TMU master e pelo menos uma TMU slave num sistema DTM em CAN. Um dos pontos fulcrais nesta rede é a necessidade da TMU Slave possuir os mesmos dados que a TMU Master. Isto implica a existência de um método que garanta a consistência de dados replicados. Quando ocorre um sincronismo dos dados entre o *slave* e o *master*, o mecanismo de consistência de dados replicados deverá ocupar os canais de comunicação o menos possível.

Neste trabalho a arquitectura interna da TMU é analisada e é apresentada uma proposta para uma implementação em FPGA. Vários problemas relacionados com a gestão dos canais de comunicação são levantados e solucionados. O principal tema desta dissertação é o estudo de um módulo interno da TMU que executa um mecanismo de consistência de dados replicados. Um algoritmo de pesquisa em árvore em conjunto com o cálculo de CRC de porções de informação é utilizado para comparar dados entre o TMU master e slave. Esta técnica demonstra a eficiência em termos de taxa de ocupação dos canais de comunicação quando comparado com a técnica de reenvio de todos os dados.

A implementação em FPGA deste módulo, denominado Distributable Table Content Consistency Checker – DT3C, demonstra a viabilidade da solução em hardware. A simulação, em Matlab, do algoritmo de pesquisa em árvore permite verificar as vantagens e limites.

keywords

Fieldbus redundancy and topology, dynamic topology management in CAN, data replication and consistency.

abstract

The dependability of a fieldbus such as the Controller Area Network can be improved by adapting redundancy, improving its topology, detecting and isolating failures. The Dynamic Topology Management (DTM) in CAN is a solution that makes extensive use of redundancy and also of failure detection and isolation. This solution is composed by two main elements: The Topology Management Unit – TMU and the Network Switch Unit – NSU. The TMU is responsible for managing the redundant communication medium and guaranteeing the access of each NSU to at least one communication medium. There is also redundancy at the TMU level, which signifies that there is always one TMU master and at least one TMU slave. One of the key points in this network is the ability of the TMU slave to have the exact same data as the TMU master. This implies that there must be a rigorous consistency scheme of replicated data. Moreover, the consistency scheme should occupy as less as possible the communication medium when a synchronization between the TMU master and TMU slave is scheduled.

I analyze and propose a TMU internal architecture for FPGA based implementation. Various issues related to the redundancy management of the communication medium are analyzed. The main focus of this work is a simple replicated data consistency mechanism module that is present inside the TMU. A tree search algorithm combined with cyclic redundancy check calculations of portions of data is used to compare data between the TMU slave and master. This technique proves to be bandwidth efficient when compared to the option of resending all the data from the TMU master to the slave.

The implementation in FPGA of this module, denominated Distributable Table Content Consistency Checker - DT3C, demonstrates that the proposed solution is achievable in hardware. Furthermore, a Matlab simulation of the tree search algorithm reveals the efficiency and limitations of the DT3C.

Table of Contents

List of Figures

List of Tables

1 Introduction

Fieldbus is an industrial network system for real-time distributed control. Fieldbus works on a network structure which typically allows daisy-chain, star, ring, branch, and tree network topologies. The use of fieldbus technology has encountered numerous applications and is well established in the automotive industry, chemical industry, shop-floor control and robotics. There are various types of fieldbuses present in the market such as CAN, FIP, PROFIBUS and others. Although each of these fieldbuses have different characteristics and are intended for different types of scenarios, they all depend on the availability of the network infrastructure (physical layer). The availability of the network infrastructure is especially important in safety critical applications as in the automotive industry.

The Controller Area Network, CAN [1], [2] fieldbus has been predominant in this area (ABS, ESP, etc). CAN is a multi-master broadcast serial bus standard. Each node is able to send and receive messages, but not simultaneously: a message (consisting primarily of an ID — usually chosen to identify the message-type/sender — and up to eight message bytes) is transmitted serially onto the bus, one bit after another — this signal-pattern codes the message (in NRZ) and is sensed by all nodes. The devices that are connected by a CAN network are typically sensors, actuators and control devices. A CAN message never reaches these devices directly, but instead a host-processor and a CAN Controller is needed between these devices and the bus. If the bus is free, any node may begin to transmit. If two or more nodes begin sending messages at the same time, the message with the more dominant ID (which has more dominant bits i.e. bit 0) will overwrite other nodes' less dominant IDs, so that eventually (after this arbitration on the ID) only the dominant message remains and is received by all nodes.

The CAN bus topology is based on a single medium, which imposes a problem for dependable systems. In order to reduce medium faults and increase dependability, a medium redundancy mechanism for CAN must be adopted. The importance of reliability, performance and scalability in a distributed system is increasing and the demands for such requirements are becoming crucial. This means that alternative bus solutions should be considered for medium redundancy in CAN networks.

1.1 Scope

In [3], a solution is described which consists in dynamically managing a redundant CAN bus topology while granting, in a transparent manner (at the node application level), extra bandwidth to each node. The purpose of this thesis is to examine the main issues in dynamically managing a redundant CAN fieldbus. Although, this dissertation shall focus and analyze the management component proposed in [3], the Topology Management Unit (TMU), the main objective will be related with the data consistency between the TMUs. One form of increasing reliability and performance is to use replicated data in a distributive system. The replicated data will be present in each TMU and for this reason it is very important that there is no data mismatch between the various TMUs. Special attention is given to the implementation of a data consistency scheme in FPGA. This implementation is done to demonstrate the validity of the proposed solution and focus only on the CAN fieldbus. The following goals are proposed in this dissertation:

1. Analyze the state-of-the-art CAN bus solutions which increase resilience;
2. Analyze and define the internal functionalities of the TMU;
3. Define a consistency scheme between TMU Master and Slave;
4. Simulate the consistency scheme in Matlab;
5. Implement the consistency scheme in a FPGA;

1.2 Organization

This dissertation is organized into the following sections:

Chapter 2 – Begins by describing dependability and classifying its attributes. Various CAN network solutions which make use of redundancy, different topologies, detection and isolation are presented and compared between each other.

Chapter 3 – A brief description is given relatively to two forms of networks, which use a specific technique of increasing dependability by dynamically managing the network resources and are designated as Dynamic Topology Management Networks.

Chapter 4 – This chapter focuses on the description of the TMU functionalities. Problems and their possible solution are presented for the following issues: Fault detection (medium and nodes), Communication medium redundancy, TMU redundancy and NSU.

Chapter 5 – The concepts of a consistent data replication method is discussed in this chapter. All procedures involved in this algorithm are explained in great detail.

Chapter 6 – Matlab simulations are used as proof of concept for the consistent data replication method described in chapter 5. An FPGA implementation in VHDL of the consistent data replication is revealed as an experimental demonstration of the concepts described in the previous chapter. Simulation and test cases results are also presented in this chapter.

Chapter 7 – Conclusion and analysis of the results obtained in this dissertation and direction for future research.

2 System Dependability

A dependable system is one that can withstand a number of sub-system and components failures while continuing to operate normally. This can be achieved by installing additional equipment, together with careful design to eliminate single points of failure and planned maintenance. A balanced approach is essential. Adding redundancy is straightforward but it is expensive and must be done after proper consideration of the costs and benefits. A resilient system requires the intelligent disposition of reliable component/unit in a redundant topology. So, increasing the reliability by introducing redundancy into a system, translates into a higher dependability of a system. The essential elements of a dependable system are [4]: Availability, Reliability, Safety, Confidentiality, Integrity and Maintainability. Making use of different topologies and redundancy schemes may not be enough to improve a system's dependability. There should be implemented additional mechanisms to detect and most importantly isolate faults from the rest of the system.

In this chapter, we will focus on redundancy, topology and fault detection/isolation in fieldbuses. The next sub chapters define and classify each element to improve resilience and categorize existing CAN fieldbus solutions.

2.1 Redundancy

As mentioned in the introduction, the use of redundancy has increased over the past years in order to fulfill customer's demands. The reason is that redundancy improves a system reliability and availability, thus improving its continuity, readiness for correct service and avoiding frequent and severe failures [5]. Redundancy is the replication of critical components of a system with the intention of increasing its reliability. Redundancy can be categorized in types and levels. In a system, there can be various levels of redundancy. For instance, in a fieldbus there can be redundancy at the node level, which means that there will be more than one node with the same functionality in the system. Inside each node, there can even be redundancy at the software level. Another popular solution is granting redundancy at the physical level, where each node can access more than one communication medium. There are also various types of redundancy:

- **Standby redundancy -** Standby redundancy means that an alternative means of performing the function is provided but is inoperative until needed. It is switched on upon failure of the primary means of performing the function. The disadvantage of standby redundancy is that there is an inevitable period of disruption between the failure occurring and the redundant unit being brought into use. Such schemes are rarely satisfactory for critical systems. *REDCAN* [8] makes use of this type of redundancy.
- **Active or parallel redundancy -** In active or parallel redundancy all redundant units are operating simultaneously rather than being switched on when needed. The most obvious approach is to use two components, each capable of executing the same tasks so that, if one should fail, the other will take over - this is referred to as 1+1 redundancy. Because there is no interruption, active redundancy is suitable for computer installations. The use of mirrored disks in a server computer is an example of 1+1 redundancy. An example of this type of redundancy can be seen in *DTM* in CAN [3].
- **N+1 redundancy -** An alternative approach is to split the tasks among a number of units, each capable of executing only a fraction of the tasks, and provide just one additional redundant unit - this is referred to as N+1 redundancy. N+1 redundancy can be cheaper to implement than 1+1, and is more flexible. A RAID 5 disk system in a server computer is an example of N+1 redundancy. Additional error correction data is added to the original data and then logically dispersed over a number of disks. If one disk fails, all the data can be recovered in real time from the remaining disks and the failed disk contents reconstructed. There is currently no CAN solution which uses this type of redundancy.

2.2 Topology

The topology of a fieldbus or network refers to a virtual communication structure between all the devices. Each topology offers different characteristics and can be categorized into the following basic types:

- **Bus Topology -** Bus networks use a common medium to connect all devices, this means that a single cable connects to all devices. A device wanting to communicate with another device on the network sends a broadcast message onto the communication medium (all devices see), but only the intended device actually accepts and processes the message. The standard CAN network [1], [2] uses a bus topology. Disadvantages are the maximum number of devices, limited bandwidth and if the backbone cable fails, the entire network effectively becomes unusable.
- **Ring Topology -** In a ring network, every device has exactly two neighbors for communication purposes. All messages travel through a ring in the same direction, either clockwise or counterclockwise. A failure in any cable or device breaks the loop and can take down the entire network. *REDCAN* [8] is a typical ring network.
- **Star Topology -** A star network features a central connection point called a hub (switch, router). Compared to the bus topology, a star network generally requires more cable, but a failure in any star network cable will only take down the devices network access and not the entire network. If the hub fails, however, the entire network also fails. An example of this type of topology is *CANcentrate* [9].
- **Tree Topology -** Tree topologies integrate multiple star topologies together onto a bus. In its simplest form, only hub devices connect directly to the tree bus and each hub functions as the “root” of a tree of devices. This bus/star hybrid approach supports future expandability of the network much better than a bus (limited in the number of devices due to the broadcast traffic it generates) or a star (limited by the number of hub connection points) alone. *ReCANcentrate [10]* can be considered an example of a mixture of a tree and star topology.
- **Mesh Topology -** Mesh topologies involve the concept of routes. Unlike each of the previous topologies, messages sent on a mesh network can take any of several possible paths from source to destination. (Recall that even in a ring, although two cable paths exist, messages can only travel in one direction). No CAN solution presents this type of topology probably due to the complexity involved in adapting the CAN standard.

More complex networks can be built as hybrids of two or more of the above basic topologies.

2.3 Fault Detection and Isolation

Most protocols have good dependability properties, but there are still some failures that can have serious consequences and therefore special care should be taken. *Laprie* [4] characterizes a failure as a deviation of the delivered service from the correct service. A failure can cause a fault in an entity, which in turn can generate errors. The problem is that errors can cause other failures, which cause faults and generate errors. This cycle (failure -> fault -> error -> failure ...) can continue to run until the system breaks down.

Depending on the severity, failures can be classified as local or as global. Global failures are the most severe and unwanted because they immediately block an entire system. A global failure in a CAN bus can happen if the communication media is stuck at dominant. In this situation the entire CAN system would not be operational. Local failures are less severe but should be prevented and taken seriously, because it can spread and ultimately cause a global failure. An example is when a node suffers from a failure that stimulates successive errors and as a final result it sets its bus output to dominant state. Bus guardians help prevent local failures from spreading and becoming global failures. A bus guardian is a device that is physically located between the node and bus. It monitors a nodes bus interface and isolates the node in case it detects incorrect behavior.

In a fieldbus, a failure can also be categorized as node or media failure. This will be described in more detail in chapter 4.3.1.

2.4 CAN Fieldbus Solutions – State of the Art

As in all systems, several solutions have been offered over the years to increase fieldbus resilience. It is interesting to verify that some solutions focus only on the topologies while others focus on the redundancy to improve the dependability of a system. There are still solutions which make use of the best of all worlds (redundancy, topology and fault isolation). These solutions are summarized and classified:

Increasing Dependability via Fault Detection/Isolation

An Analyzable Bus-Guardian for Event-Triggered Communication [6] uses only a bus guardian on each node to confine erroneous transmissions and enforce fail silent behavior. This solution does not make use of redundancy or topology properties to increase dependability.

Increasing Dependability via Redundancy

Rufino *et. al.* [7] proposed the *Columbus' Egg Idea* which essentially uses media redundancy. Each node can detect and ignore a faulty media by comparing the data received on the various buses. Fault isolation due to incorrect node behavior is not possible and therefore can cause a global failure. Spatial proximity is also a property that does not contribute to increase the systems dependability.

Increasing Dependability via Topology

Spatial proximity of entities in a system can become a problem when dealing with hazardous environments. For example, in a bus topology, errors (grounded wires, loose connectors, faulty transceivers and cable partition) propagate through the common transmission medium, possibly affecting the whole system. One form of solving the spatial proximity problem is making use of a different topology.

An approach to use CAN with an alternative topology is *REDCAN* [8]. The *REDCAN* bus topology is based on a ring architecture, where each node has a reconfiguration switch that allows the isolation of the defective bus segment between each node. The downside of this solution is the recovery time from a defective bus segment situation.

CANcentrate [9] adopts a star topology. In the center of the star is an active hub that has the ability to broadcast all receiving messages. The hub also can detect faults (stuck-at node, bit-flipping, medium shorted and partitioned) and isolate each node. The bus guardian located in the hub has an advantage relatively to a bus guardian in a bus topology which is the fact that it doesn't suffer from spatial proximity with each node. Although this solution makes use of two communications medium, one for RX and the other for TX, this cannot be considered redundancy.

Other star topology solutions such as the *Passive Star* [11] and *Active Star* [12] only address the spatial proximity issue and are incapable of isolating faulty nodes or branches. The *starCAN* [11] focuses only on the network performance and even presents a decrease in the systems dependability. This last solution demonstrates that changing topology does not automatically denote a higher dependable system.

Increasing Dependability via Redundancy and Fault Detection/Isolation

The *DTM* in CAN, [3] takes advantage of redundancy at all levels while maintaining a bus topology. There is bus, TMU, NSU and node redundancy. The TMU has a bus guardian which verifies the state of each bus. Failure in any one of the buses is detected by the TMU master which reconfigures all NSUs to use the appropriate buses. Each NSU acts as a bus guardian and connects N nodes (which can be redundant) to the M buses. There is still a possibility to have the same nodes on different NSUs, thus guaranteeing redundancy of node and NSU. Fault isolation is done on the media and node level.

Increasing Dependability via Redundancy, Topology and Fault Detection/Isolation

ReCANcentrate [10] differs from its predecessor (CANcentrate) because it uses hub redundancy. Using communication media redundancy in a star topology can be a

challenge and the cable length will be double the size. No solution has yet made use of topology, redundancy (at all levels) and fault detection/isolation while maintaining transparency to the node application. The increase in dependability may not be worth the effort.

The following table classifies each of the solutions described above in terms of topology, redundancy, fault detection, isolation and other important factors.

CAN Solutions	CAN Compliant	Topology	Redundancy Type/Level	Fault Detection	Fault Isolation	Weakness	Bandwidth increase
An Analyzable Bus-Guardian for Event-Triggered Communication	Yes	Bus	No	Yes	Node level	Spatial proximity	No
Passive Star From Cena	No	Star	No	No	No	Strong limitations (hardware)	No
Active Star from Rucks	Yes	Star	No	No	No	No fault detection or isolation	No
StarCAN	No	Star	No	No	No	Not dependable	Yes
Columbus' Egg Idea	Yes	Bus	Active Communication medium	Yes	No	Spatial proximity	No
RedCAN	Yes	Ring	Standby	Yes	Yes	Recovery Time	No
CANcentrate	Yes	Star	No	Yes	Node and media level	Hub is a single point of failure.	No
ReCANcentrate	Yes	Star	Active Hub	Yes	Node, media and Hub level	Cable Length	No
Dynamic Topology Management	Yes	Bus	Standby Communication medium, TMU and Node	Yes	Node and media level	Recovery time may be an issue	Yes

Table 1 – CAN Solutions for Dependability Improvement

3 Dynamic Topology Management in CAN

The use of redundant buses and different topologies in CAN has already been proposed by some authors mainly to deal with fault tolerance issues. A small overview of CAN solutions that improve fault tolerances issues can be found in chapter 2 - System Dependability. However, a few solutions have emerged in the last years with the idea of real time management of redundant CAN networks, thus reallocating resources on the event of a communication media failure or the need to increase bandwidth. The basic idea behind these architectures is efficient resource managing.

An architecture that improves the bandwidth and that implements redundancy is proposed in [15] and [16]. This architecture uses several CAN fieldbuses (at least two) which operate as parallel transmission channels that can be used either to improve the bandwidth by transmitting different traffic in different channels or to promote redundancy by transmitting the same messages in more than one channel, as shown in Figure 1 – **FTT-CAN with Master and Bus Replication**. The FTT-CAN master node does the control of the transmission. Among other properties, this architecture inherits the dispatching flexibility of FTT, thus enabling on-line changes in the traffic conveyed in the channels. This is useful to fulfill application requirements or to react to bus failures leading the system to a degraded operational state, without compromising safety. It should be noted that it is not necessary that all the nodes connect to all the buses. This enables the division of nodes in functionality clusters and the use of simpler nodes, with just one CAN controller, for example exactly identical to the ones used presently.

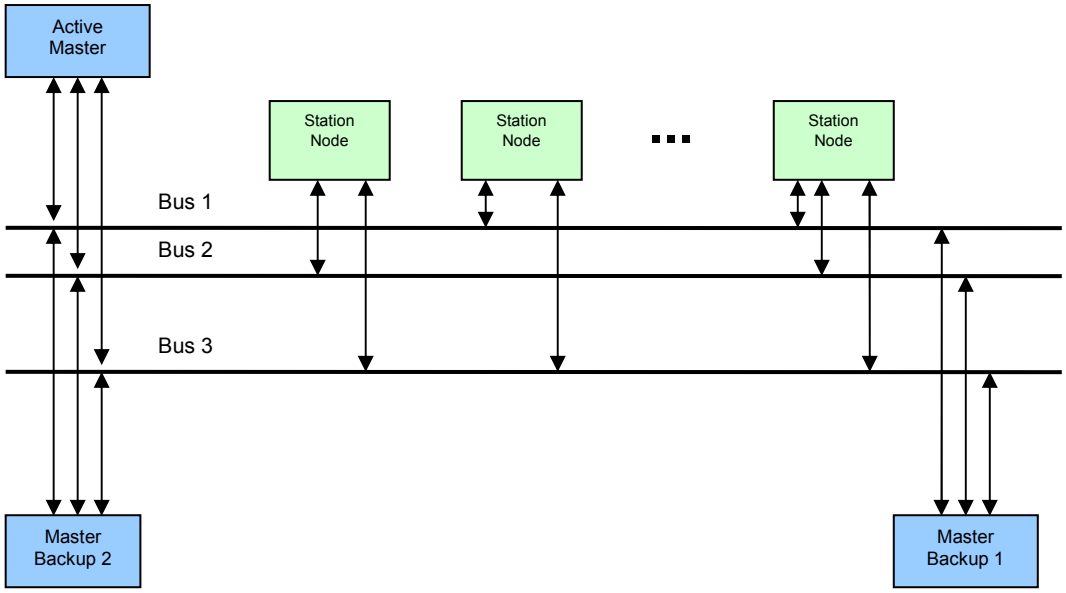

Figure 1 – FTT-CAN with Master and Bus Replication

The master node, which is a centralized node with the complete control of the bus, can schedule the traffic in an efficient way in the same bus or in different buses. In FTT-CAN, the master node schedules the traffic in the bus using the data stored in a table, the Synchronous Requirements Table (SRT). In the proposed architecture the master node does the scheduling recurring also to the SRT, however this table can have, in some of the previewed operational modes, an indication of the bus (or buses, when using redundant messages) that must be used to transmit the messages. The Master manages then more than one bus at the same time. The proposed system also includes replicated masters, adopting a leader follower behavior. The system only has a single active master at each time, being all the others backup masters. In case of an error in the active master, one backup master will become active since the previous active will be stopped (fail silent). FTT-CAN slots the bus time in consecutive Elementary Cycles (ECs) with fixed duration. All nodes are synchronized at the start of each EC by the reception of a particular message

known as an EC Trigger Message. The master nodes are located at the end of the buses and the number of backup masters at one end of the buses equals the number of backup masters at the opposite end. This facilitates the bus error detection since one Trigger Message omission can be easily detected by the master located in the opposite end of the bus. In this way, if a Trigger Message is omitted the backup master located at the opposite end of the bus will inform the active master of the error. The active master, if not crashed, could then re-schedule the traffic to the non faulty buses.

Another proposed network [3] is based on a set of components, protocols and a topology architecture that is capable of granting the availability of the network infrastructure to all nodes, increase the total available bandwidth, detect and isolate the defective buses during operation, as shown in Figure 2 **– DTM Fieldbus Topology**.

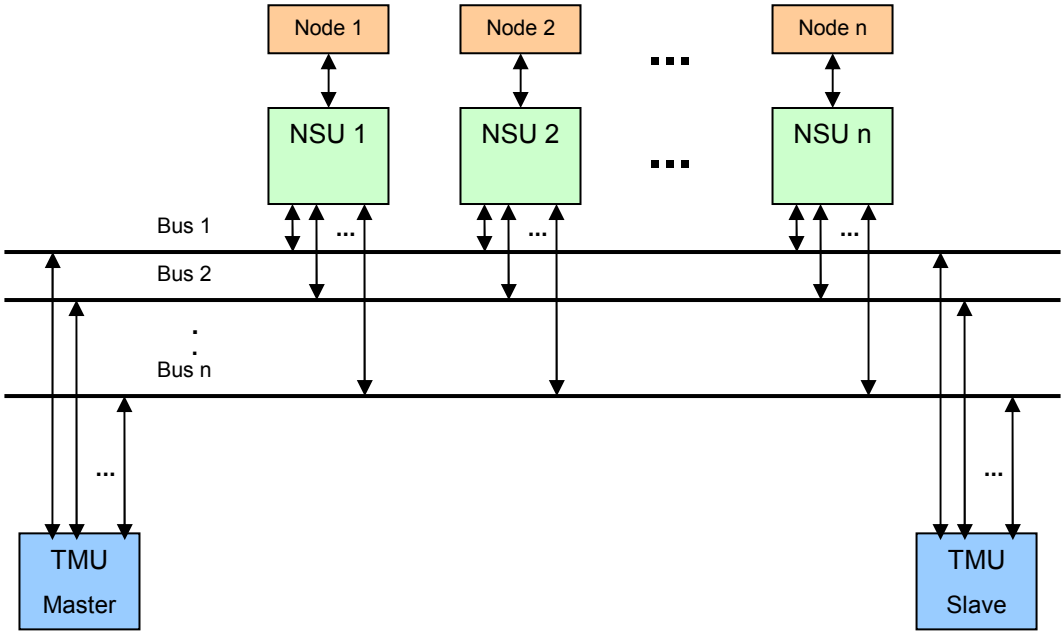

Figure 2 – DTM Fieldbus Topology

The network infrastructure is based on redundancy, meaning that each node has access to various physical mediums via a Network Switch Unit, NSU. The Topology Manager Unit (TMU) specifies the physical medium(s) which is/are active for each NSU. The TMU master monitors and detects faults on the physical medium. Each time a fault is detected on a physical medium, the TMU reconfigures each NSU, thus isolating the faulty media. According to the node's bandwidth needs, the TMU is also responsible in giving an extra bandwidth to each node by assigning more than one media. However if there is a safety critical request then the TMU can configure the NSU to access and dispatch the same message over more than one communication media. In order to guaranty a higher fault tolerance there are at least two TMUs located on each end of the buses. As the previous solution, the number of TMUs on one end equals the same number of TMUs on the opposite end. One of the TMU will be a master and in an active-active state, while the other TMU will be a slave and in an active-standby state. This means that if the master's state changes to a fault state, then the TMU slave shall change its state from active-standby to active-active, assuming control over the buses and NSU's.

Both solutions are reasonably similar in terms of dependability, but with some significant differences in terms of implementation:

- The FTT-CAN with master and bus replication is a software based solution. The software lower layers of each node must be adapted in order to integrate them into the solution. The master nodes are no more than simple nodes with specific master management software.

- The DTM in CAN network is a hardware solution. This means that all software running on each node does not need any intervention in order to integrate them into the DTM. The TMU and NSU are specific hardware implementations in FPGA/ASIC.

The FTT-CAN solution presents a huge disadvantage in upgrading a standard CAN network. Explaining to a potential client that it will be necessary to update/change the stable software in each node can pose a great obstacle. The DTM has this advantage, since the upgrade is totally transparent to each node. However, because the TMU and NSU are not COTS, the financial investment will be greater than the FTT-CAN solution. If there is a need for the dispatching flexibility of FTT or for a new system, then the FTT-CAN solution is the optimal choice. The TMU in the DTM solution will be studied in detail in the following chapters.

4 Study of a TMU Architecture

In this chapter the TMU behavior and protocol will be described. Also, a description of the functional architecture of the TMU will be given and the main form of fault detection in the communication media will be identified. The communication between the redundant TMUs and NSUs is characterized in terms of configuration, verification and synchronization messages. Finally, a TMU hardware implementation and its integration in the DTM network will be proposed.

4.1 Functional Architecture

The Topology Manager Unit – TMU plays an important role in the Dynamic Topology Management network. The TMU will act as a guardian for all transmission media and allocate each media to a specific number of Network Switch Units – NSU. The TMU has also the capability of isolating a communication media. The TMU proposed by [3] divides each functionality into the following blocks as shown in Figure 3 – **TMU Internal Functionalities** and described below:

- Topology Reconfigurator: decides which are the appropriate buses for each NSU. This decision is based on the information given from the Bus Failure Detector. The actual decision is stored and read from the Reconfiguration Table;
- Bus Failure Detector: detects various types of faults (stuck at, partition, etc) on each bus and informs the Topology Reconfigurator;
- Reconfiguration Manager: responsible for carrying out the topology reconfiguration on each NSU and TMU Slaves;
- Topology Table: stored information that contains the position of each NSU;
- Reconfiguration Table: stored information which stipulates the procedures for every possible bus failure;
- NSU Tables: contains bus switch information for each NSU present in the system. The bus switch information defines the buses which are used or unused by each NSU;
- External Configuration Interface: allows to upload the tables to the TMU;
- QoS Indicators Interface: gives status and system information which can be used for statistical purposes.

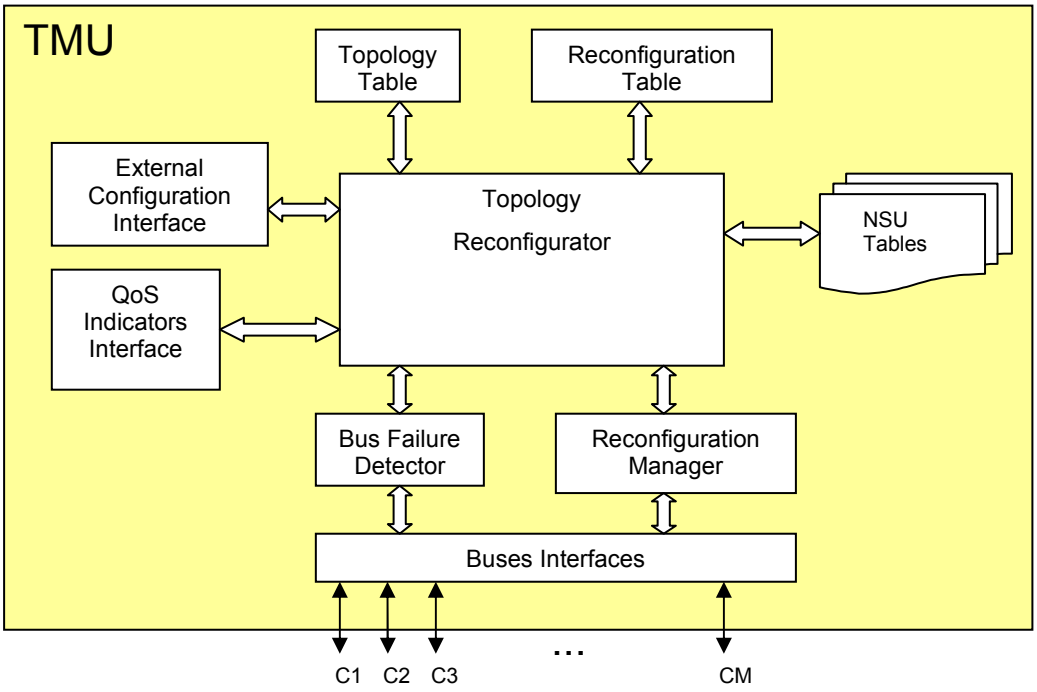

Figure 3 – TMU Internal Functionalities

The components proposed by [3] and shown in Figure 3 – **TMU Internal Functionalities**, may not correspond directly to an implementation solution. However their functionalities should be present in the TMU.

4.2 Problems related with DTM Redundancy

The use of redundancy in systems normally is not straightforward and there are plenty of issues that have to be solved. This applies to the DTM. The main concern goes to the TMU who is responsible for the network management. There are many questions that arise and must be answered:

1. When should the TMU Slave take over the system management? What are the conditions that will trigger this event?
2. How can the TMU Master detect faults?
3. Which types of faults are detected?
4. Keeping in mind the availability of the communication medium, how can the TMU master communicate with the TMU slave and NSU without occupying the majority of the bandwidth?
5. Should TMU Master transmit on all medium simultaneously?
6. Should TMU Master expect to receive simultaneously the same data on all medium? If the TMU receives different data, which data should it choose as valid?
7. How can one guarantee that there is data consistency between the TMU Master and the TMU Slave in an efficient way?

Answers and solutions to questions 1 to 6 are given in this chapter (4). While the main issue (question 7), consistent data replication will be addressed in chapter 5.

4.3 Fault Handling

In order to specify a fault detection mechanism appropriate for the TMU it is necessary to identify which situations are considered faults in the DTM system. Therefore a fault model shall be defined. It is also important to distinguish fault containment and error containment in the TMU. In [4], fault containment is defined as the ability to limit the immediate impact of a single fault to a defined region within a system, while error containment is the ability to avoid the propagation of the consequences of a fault (error) out of the fault containment region. Faults occurring in the node are the responsibility of the NSU and therefore out of the scope of this thesis. So the main interests are the faults occurring in the communication medium and in the TMU itself.

4.3.1 Fault Model

A CAN network is made up of at least two nodes and a channel, whereas the node composes the following components:

- Host controller: this component runs the application software;
- Communication controller: this component sends and receives data encapsulated in the CAN frame;
- Transceiver: this component is responsible for converting the CAN frame to and from the channel voltage levels.

The faults that occur on the CAN network can be originated by problems on the medium or on the node.

The CAN transmission medium is usually a two-wire differential line. The CAN physical layer specified in the ISO Standard [2] is tolerant against some of the transmission medium failures by switching from the normal two-wire differential operation to a single-wire mode. This switch-over bus operation happens in the presence of one of the following failures:

- One-wire interruption;
- One-wire short-circuit either to power or to ground;
- Two-wire short-circuit.

CAN medium interfaces which have switch-over functions are commercially available. However, these failures shall be considered in the fault model in order to support all interfaces. So, besides the previous faults the CAN transmission medium may suffer from:

- Bus partition: Simultaneous interruption of both bus line wires;
- Bus termination failure: lack of correct bus termination leads to signal reflections which can compromise communication integrity. A solution can be achieved by taking into account the extra time needed for bus signal stabilization;
- Stuck-at-recessive: Stopping to deliver other than recessive symbols. This happens when both transmission wires are short-circuited to power;
- Stuck-at-dominant: Stopping to deliver other than dominant symbols. This happens when both transmission wires are short-circuited to ground;
- Babbling idiot fault: consists in transmitting information in an erroneous form without restrictions in the time or value domain. An example is a bad welding of the medium connector, which can generate random bit values. Some babbling idiot faults will not cause a global communications fault but will decrease the overall performance, while others can permanently disable any correct communication on the medium;
- Massive transient disturbances: caused by electromagnetic emission (EMI), which can lead to a temporary loss of communication.

The nodes itself can also be a source of failure. These faults may be:

- Stuck-at-recessive or dominant: damaged node issuing a constant bit value – 0 or 1. In this case, only the stuck-at-dominant fault causes global communication failure since the CAN physical layer is equivalent to a logic AND of every node's contribution.
- Babbling idiot fault: normally this fault is characterized by a node entering an infinite loop, sending continuously high priority messages and therefore occupying the transmission medium. The most successful solution [14] consists in putting a bus guardian between the medium and the node. This bus guardian will have the task in analyzing the messages and silencing the node whenever a babbling fault occurs.
- Masquerading failures: occurs if an erroneous node assumes the identity of another node. These faults, specified in [17], are difficult to diagnose if the receiver has no knowledge about the identity of the sender of a message. This is the case of the CAN network.

The communication failures caused by problems on the nodes (stuck-at, babbling idiot or masquerading failures) are best solved by the NSU, therefore the TMU's fault model will only take into account the failures on the transmission medium, as shown in Figure 4 and resumed in Table 2.

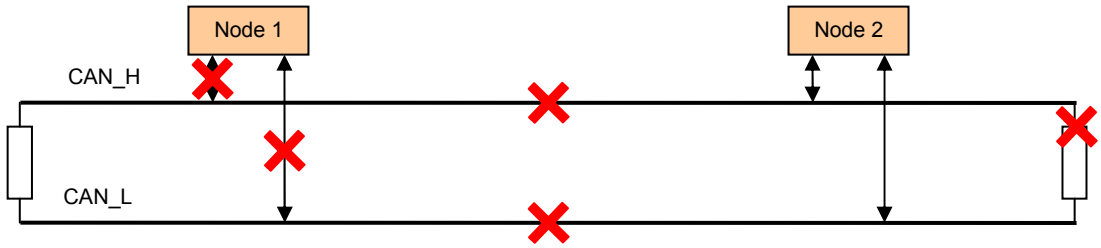

Figure 4 – Failures on the CAN Network

	Failure Mode					
	Bus Partition	Bus Termination Failure	Stuck-at-recessive	Stuck-at-dominant	Babbling idiot	Massive transient disturbances
Failure effect	Possible node isolation	Signal degradation	Total loss	Total loss	Total loss of communication if fault is permanent	Transient loss of communication while disturbance is present
Possible fault handling	Detect and use redundant bus	Redimension the CAN bus timing parameters	Detect and use redundant bus	Detect and use redundant bus	Detect and use redundant bus	Detect and use redundant bus. Possible to resynchronize previous bus after no EMI

Table 2 – Fault modes and their possible handling

4.3.2 Fault Detection Mechanisms

There are various methods for fault detection on the physical layer. Some of these methods are still under research and need improving while others are well known and adopted by some communication systems. These methods will be briefly presented, pointing out their main advantages and disadvantages.

Fault Detection in Fieldbuses with Time Domain Reflectometry

This method [18] consists in sending a pulse on the fieldbus medium and analyzing afterwards the pulse reflections. In order to analyze the pulse reflections, a reference image has to be created from a simulation. Another time domain reflectometry - TDR image is taken before fieldbus operation and used in combination with the simulated image. Only then, TDR can be applied during operation for fault detection.

Advantages:

i. This method can be used on any fieldbus;

ii. During fieldbus operation (packet transmission), the TDR analysis can be done simultaneously;

iii. Real time detection.

Disadvantages:

i. Complexity, both in implementation (hardware) and in reflection pattern recognition;

ii. Not mature;

iii. Each time fieldbus topology (node insertion / removal, cable length) changes, a new reference image has to be created.

Neuro-Fuzzy Fault Detection Approach

A fault diagnose and identification – FDI approach [19], which consists on a neuro-fuzzy system for fault detection is coupled to the fieldbus communication media. The idea is to use a model-based approach which compares the output signals of the model with the real values measured from the process, thereby generating residuals, which are fault indicators. The model method used is the neuro fuzzy which monitors the communication media with quantitative and qualitative modeling methodologies, and achieves a learning system with transparent knowledge.

Advantages:

i. Fast and robust implementation;

ii. Learning ability;

iii. Real time detection.

Disadvantages:

i. Exponential growth of the network structure when the dimension of input-output space increases or the number of fuzzy partition increases;

ii. The complexity of the N-F structure causes the drastic growth of the computational cost during the learning process;

iii. Knowledge of methods to reduce the N-F network structure.

Guardian Approach

The use of guardians is another fault detection method. The guardian can be located on the node [20] itself or on a separate entity in the fieldbus. The neuro-fuzzy fault detection

can be considered as a guardian. Guardians can also be distinguished as guardians with timing failure detection and/or value failure detection. In a system with a time triggered protocol there can be simultaneously timing failure detection and value failure detection. The guardian [20] monitors the temporal access pattern of the controller to the communication media and terminates the controller operation in case of a timing violation. In a system with an event triggered protocol, the bus guardian cannot determine temporal patterns and therefore can only analyze the communication media in terms of protocol violations (i.e. in CAN there is a maximum of 6 dominant symbols allowed).

Advantages:

i. Real time detection;

ii. Efficient detection (in time triggered networks).

Disadvantages:

i. Complex to implement in certain architectures such as event triggered networks.

4.3.3 Adopted Solution for TMU

The purpose of this subchapter is to study the best form of detecting faults on the medium. In the Dynamic Topology Management (DTM) network, besides the medium fault detection, the TMU has the necessity to:

- Manage and configure each NSU's access to the communication medium;
- Manage each redundant TMU Slave.

As seen previously, there are various mechanisms or techniques in detecting a fault in the bus medium. A small experiment can be found in appendix A1 - **CAN Communication Media Experiment**, analyzes the best form of detecting faults on the medium. Solution 3 is the simplest for various reasons such as:

- The use of a dedicated communication medium just for the TMU configuration and management raises the problem of extra resources and the implications of its own failure. It would also mean that extra hardware would be necessary to detect medium faults on the main communication medium.
- If the TMU uses the preexistent communication medium, then standard CAN messages shall be used for the TMU medium fault detection and redundancy management and configuration. Thus, the communication between TMUs can be used to verify media faults (partitions, stuck at recessive, etc) using a *ping/ping acknowledge* strategy and has the advantage of no need of extra hardware.

Solutions 1 and 2 in appendix A1 - **CAN Communication Media Experiment** can be of interest for future research and improvement of the DTM in terms of bandwidth efficiency.

To conclude: The TMU will use solution 3, thus using the communication medium for redundancy management (table dispatch and configuration), while verifying the medium status. A compromise must be achieved between the amount of data traveling between TMUs in order to keep the communication medium available for the nodes as much as possible. The fault detection mechanism will also occupy the communication medium since it will be based on a *ping/ping acknowledge* strategy between the TMU master and its slaves. Again a compromise should be made in the amount of *ping* messages dispatched in a certain time interval and because the communication medium is shared among various other entities (nodes), a less stringent time to live for each *ping/ping acknowledge* should be taken into account. The *ping/ping acknowledge* messages will also piggy back information necessary to verify if the tables are synchronized among the TMUs. All of these issues shall be discussed in the following subchapters.

4.4 TMU Redundancy Management

The TMU has to deal with the redundancy of the communication medium and with its own redundancy (TMU slaves). As explained in 4.3.3, there was an attempt to use each bus medium to exchange data between TMUs at a higher data rate (much above the 1Mbit/s), without interfering (or occupying additional bandwidth) with the normal CAN messages exchanged between nodes. As this is not the adopted solution, the communication between the TMU Master and the TMU Slave is done via standard CAN messages.

4.4.1 Communication between TMUs

The TMU master is the only entity that will initiate a communication in the DTM (excluding the normal traffic between nodes). This means that the TMU slave will only send a message upon a request from the TMU master. Communication between the TMU master and its slaves have the objective to:

a. Dispatch a table;

b. Maintain tables synchronized;

c. Verify the communication medium (bus partition or other problems).

Therefore there will be various commands to support all these functionalities. The data field of the CAN frame will contain the command and parameters. A group of commands were defined for the communication between TMUs and can be found in A2 - **TMU ARQUITECTURE**. The commands are classified either as verification (*ping* and *ping_response*) or synchronization (config) messages. The table dispatch and synchronization was identified as one of the most important issues in the TMU, therefore an entire chapter (5 - **TMU Table Synchronization**) is dedicated to this functionality.

TMU Slave configuration

The following example demonstrates the configuration of the NSU table (the same applies to the other tables and ConfigRegBank)

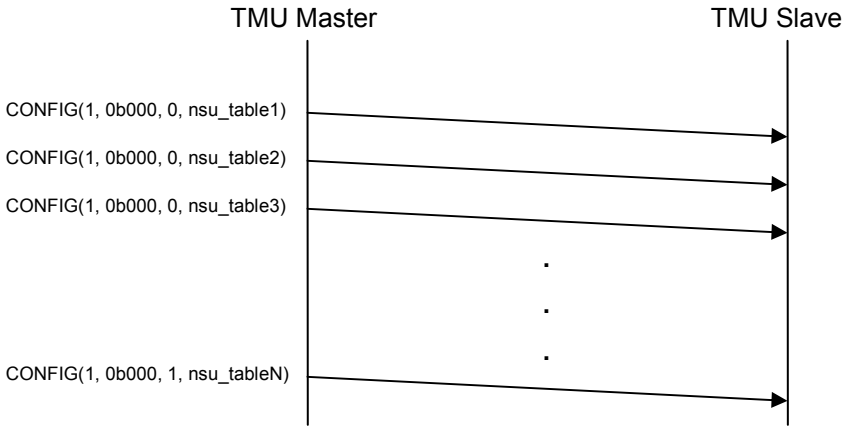

Figure 5 – TMU Configuration

The example presented in Figure 5 – **TMU Configuration** does not make use of the acknowledge request. If the acknowledge in all the config messages from the TMU master was asserted then the TMU slave would have to send an acknowledge message for each received config message. The parameters in Figure 5 – **TMU Configuration** are explained in.A2.1 - **TMU Management messages**

TMU during normal Bus operation

The following example demonstrates how the *ping/ping response* flow works. Notice that the initial CRC value is CRCX and then the NSU table is updated and the CRC value switches to CRCY. As long as there is no mismatch between the CRC values in master and slave the TMU behaves always as described in Figure 6 – **TMU Master / Slave Communication**. CRC mismatch is described in detail in chapter 5 - TMU Table Synchronization. The parameters in Figure 5 – **TMU Configuration** are explained in A2.1 - **TMU Management messages**.

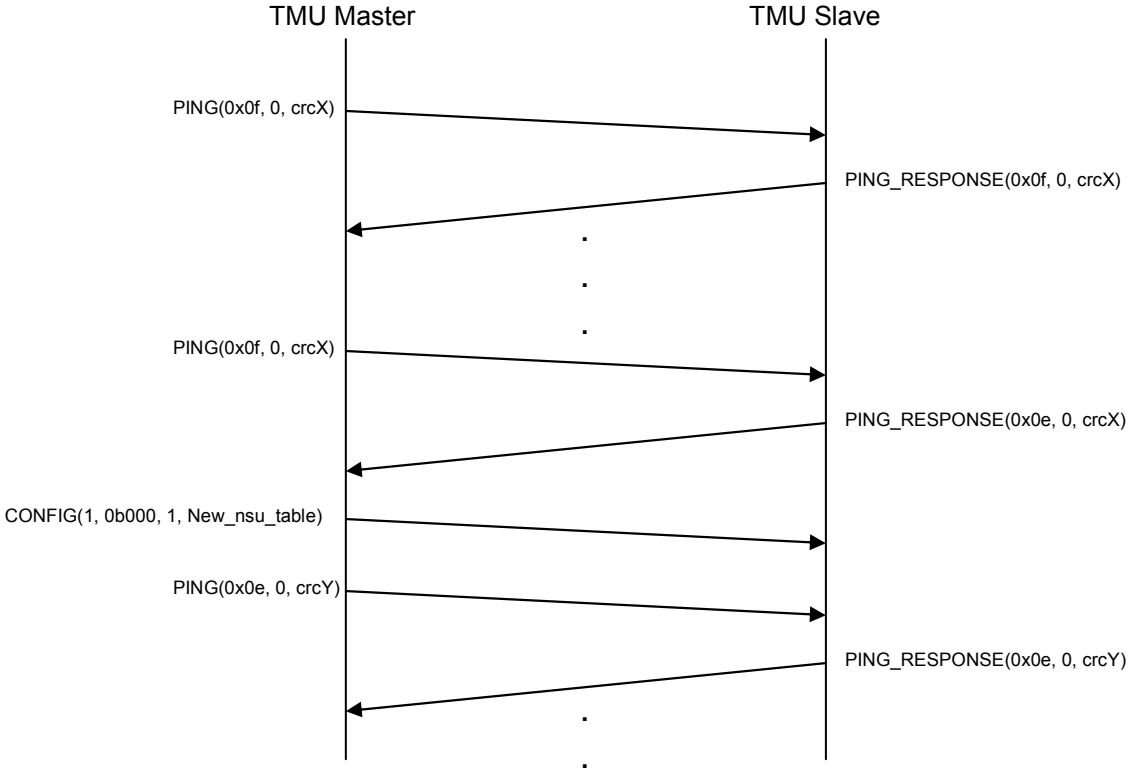

Figure 6 – TMU Master / Slave Communication

4.4.2 Medium Redundancy Management

The main objective of the TMU is to manage the medium in order to provide the NSUs continuity of medium access and even to increment the bandwidth. Fault detection is done in each TMU with the ability to verify various errors. The TMU has various media channels connected and needs to have a criterion to deal with information that is sent and received over all media channels. All of these issues are discussed below.

4.4.2.1 TMU Bus Guardian

In the TMU (master and slave), each communication media has a bus guardian. The Bus Fault Detector – BFDunit (see 4.7 - **TMU Brief Implementation Description**) is an internal block of the TMU responsible for supplying the Reconfiguration Manager - RecMan, with information about the health status of each bus. The BFDunit will monitor each CAN bus for certain types of failures that can occur. In the event of an error on one of the buses, the BFDunit will trigger a signal indicating the RecMan that the bus is no longer suitable for use. The following table shows which errors are monitored by the BFD and their respective action:

Failure	Nº of occurrences to trigger Bus Fault	Action
Bit error	3	Disable Bus; Change NSU table and Reconfigure NSUs
Stuff error	3	Disable Bus; Change NSU table and Reconfigure NSUs
CRC error	3	Disable Bus; Change NSU table and Reconfigure NSUs
Form error	3	Disable Bus; Change NSU table and Reconfigure NSUs
Ack error	3	Disable Bus; Change NSU table and Reconfigure NSUs
Stuck-at-recessive	-	Disable Bus; Change NSU table and Reconfigure NSUs
Stuck-at-dominant	-	Disable Bus; Change NSU table and Reconfigure NSUs

Table 3 – Bus Guardian

The BFDunit will have a debounce mechanism for each error on each CAN bus. This means that a fault will be reported to the RecMan after 3 consecutive failures on a CAN bus, thus avoiding and filtering out transient faults. This debouncing mechanism prevents any unnecessary actions that were a result of a bus status toggle. The number of faults which trigger an alarm to RecMan is 3. This value is only based on a theoretical best practice and could be changed in the future if further research demonstrates any advantages or enhancements. After a Bus is reported to be in a fail state, all NSUs receive new information so that the faulty bus is isolated from the system.

4.4.2.2 TMU Media Fault Detection

The TMU Master sends a *Ping* command to the TMU slave or NSU on all CAN medium (even the buses that are considered faulty) at a predetermined time interval. This action has the purpose of discovering bus partitions or any sort of bus failure described in 4.3.1 - **Fault Model** and briefly explained in Table 4 – **Media Fault Detection**.

Failure	Action
Bus Partition	Ping each NSU (described in 4.5.1.2 - Bus Partition.)
Different view between TMU Master and Slave (mismatch between *Ping / Ping_Response*)	Logical AND of all views (described below)

Table 4 – Media Fault Detection

The *Ping* command contains information about the bus status from the TMU master's point of view. The TMU slave registers on which buses it received the *Ping* command and replies back to the TMU Master with a *Ping_Response* command. The *Ping_Response* command also contains information about the bus status from the slave's point of view. When the TMU master receives a *Ping_Response* it must verify:

1. on which buses did the TMU master receive the *Ping_Response* – $View_{Ping\_Rsp}$;
2. which is the content of the *Ping_Response* (TMU Slave's bus status) – $View_{Slave}$;
3. which is the actual bus status of the TMU Master – $View_{Master}$.

Analyzing the 3 steps above, the TMU Master must react to various situations that are described in Table 5 and Table 6.

View	Step 1	Step 2	Step 3	TMU Master action
Normal situation	Status A	Status A	Status A	Maintain NSU table
Bus Failure	Status B	Status B	Status A	Change NSU table to Status B
Bus Recovery	Status A	Status A	Status B	Change NSU table to status A
Bus Status Mismatch	Status A	Status B	Status A	After 3 *Ping/Ping_Response*, change NSU table to Status B. Possible problem: • TMU Master Tx on Bus#4 or • TMU Slave Rx on Bus#4
	Status B	Status A	Status A	After 3 *Ping/Ping_Response*, change NSU table to Status B. Possible problem: • TMU Master Rx on Bus#4 or • TMU Slave Tx on Bus#4
	Status A	Status B	Status C	Total bus mismatch. Change NSU table to Status D.
	Status A	Status B	Status B	Maintain NSU table as Status B.
	Status B	Status A	Status B	Maintain NSU table as Status B.

Table 5 – Bus Status

Where for example for a system with 4 buses:

Bus Status	Bus#4	Bus#3	Bus#2	Bus#1	1 –> OK, 0 –> NOK
A	OK	OK	OK	OK	0xF
B	Not OK	OK	OK	OK	0x7
C	OK	OK	OK	Not OK	0xE
D	Not OK	OK	OK	Not OK	0x6
…	…	…	…	…	…

Table 6 – Bus Status Example

Note: Not Ok – means that the TMU Master/Slave did not receive a *Ping/Ping_Response* on the Bus.

When there is a bus status mismatch, the TMU master will disable all buses that are not receiving a *Ping* or *Ping_Response*. This means that the result between the various views is a logical AND operation:

$$\text{TMU Bus Verification}_{\text{Master}} = [\text{View}_{\text{Ping\_Rsp}} \text{ AND View}_{\text{Slave}} \text{ AND View}_{\text{Master}}]$$

4.4.2.3 Message Dispatch over the Media between TMUs

In the proposed architecture, more than one CAN bus is present. This means that it is possible to make different combinations with the transmission of verification/synchronization messages. In what concerns verification/synchronization messages, four scenarios are possible:

a. Only a verification/synchronization message is transmitted in one of the buses.

b. An identical verification/synchronization message is transmitted in every bus.

c. A verification/synchronization message is transmitted in every bus and the verification/synchronization messages can be different.

d. The transmission of verification/synchronization messages in only some of the buses.

Scenario a) does not take advantage of the redundant communication medium and is actually less favorable in a dependability point of view. It could be an advantage for bandwidth usage for the nodes. However, verification messages must be transmitted on all media in order to verify the actual bus health.

In terms of verification messages, scenario b) is mandatory although it is not favorable to bandwidth usage. This scenario will be the adopted solution for verification messages, thus informing the TMU master of the bus health.

Scenario c) will not be considered since it requires a much higher level of TMU complexity.

Finally, scenario d) may be more suitable for synchronization messages. Instead of using all communication medium to dispatch the table, which is unnecessary bandwidth waste, the solution could be to reserve certain communication medium for critical messages and use at least two communication medium for table dispatch or resynchronization messages. This will guarantee that critical information will not be delayed during a table dispatch or resynchronization.

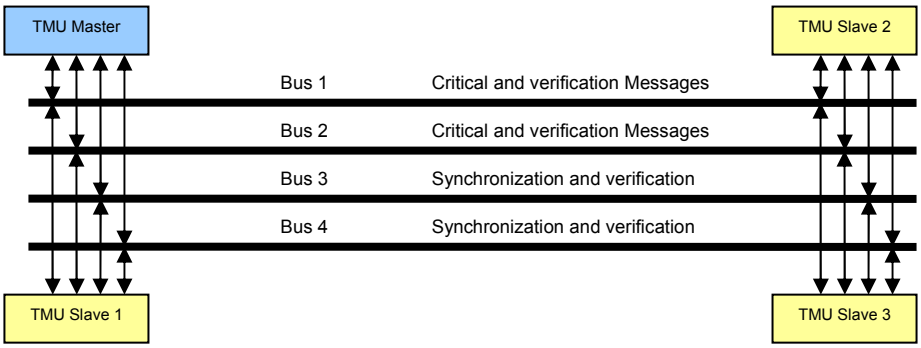

Figure 7 – Communication Medium Resource Allocation

So the adopted scenario for the DTM actually depends on the type of message that is going to be transmitted over the communication medium, as seen in Figure 7 – **Communication Medium Resource Allocation**. In the case where only two communication medium are used in the DTM, verification and synchronization messages are dispatched on both medium and therefore there will be no dedicated media for critical messages.

The rule is that the TMU master will always transmit on all communication medium for verification messages and allocate (defined in NSU table) at least two medium for synchronization messages. These messages are the system's highest priority messages. This means that special care should be taken when receiving messages, since all communication mediums are used to send the same command. As the TMU master is always responsible for initiating any type of communication (table synchronization, *ping*), it is always aware of the expected incoming message. If the incoming message is not the expected then the TMU master will simply ignore (except if it is a *ping*, which means that another TMU master is present and therefore shall go into fail mode). The content of the expected incoming message is compared between the various communication mediums. Rules decided for the message handling in the TMU Master are:

1. Always transmit on all communication medium for verification messages. At least two communication medium are assigned for synchronization messages;
2. Accept only expected incoming message type:
    a. Buffer received message on each media. Use a timeout to wait for all messages on each media;
    b. Compare message content;
    c. Use majority voting scheme if there are mismatches (empty buffers do not count, this means that no message was received on the Bus);
    d. In case of a tie for the voting scheme, discard all messages.

The situation in the TMU slave is a little bit trickier, since it never knows which kind of messages it can receive (table synchronization, *ping*). The rules for message handling in the TMU slaves are:

1. Buffer message received on each media. Use a timeout to wait for all messages on each media;
2. Compare message content;
3. Use majority voting scheme if there are mismatches (empty buffers do not count, this means that no message was received on the Bus);
4. In case of a tie for the voting scheme, discard all messages.

4.4.3 Redundancy Protocol between TMUs

There are three important issues concerning the TMUs interaction between each other: one is how the TMU slave knows when to assume control over the DTM; another situation is what happens when a TMU master receives a message that was not triggered by him; and finally there is an issue over who assumes control when there are various TMU slaves.

Master Fail Conditions

The TMU Slave has to guarantee that if the TMU Master suffers from some kind of failure, it assumes control of the system – hence the TMU Slave changes to Master mode and manages the system. The condition on which the TMU Slave decides to change to master mode is based on the following rules:

a. Lack of *ping* reception on all CAN buses for a predefined time interval;

b. Reception of a *ping* command from the TMU Master in which its view (more on this in 4.4.2.1 - **TMU Bus Guardian**) indicates no operational bus;

It will be assumed that TMUs have a fail silent behavior both in time and value domains.

Master replacement

In the DTM system, the TMUs (master and slaves) will always be located at one of the extremities of the communication media and never in the middle as shown in Figure 8 – **TMU Location**. This reduces the complexity of the TMU redundancy management and media fault detection mechanism. The system is considered fully functional when there is at least one TMU slave on the opposite extremity of the communication media from the TMU master. This means that upon the TMU master failure, the TMU Slave 1, 2 or 3 should take control over the system management. If by any chance the TMU slave 1 also becomes faulty then there will be an extremity of the communication media depleted of any functional TMU. In this situation it is not possible to determine medium faults and therefore DTM management is not possible. An alarm will be raised and service intervention will be necessary to reestablish the DTM system.

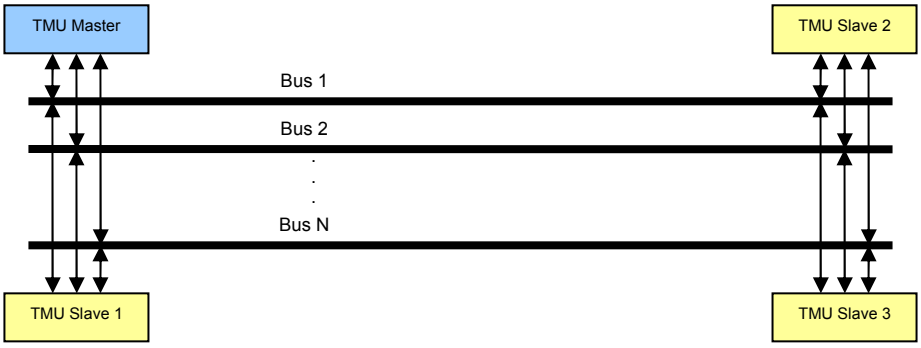

Figure 8 – TMU Location

The rule for TMU master replacement is that the slave with the highest priority ID will become master. As soon as one of the conditions for TMU master replacement is met than all TMU slaves will try to send a *ping* command on all communication medium. At this point, there could be a situation where for example TMU slave 2 sends a *ping* command first. The TMU slave 3 will verify that it has a lower priority ID and therefore will continue to behave as a slave. On the other hand, the TMU slave 1 will ignore this *ping* command since it has a higher priority and therefore will transmit a *ping* command. TMU slave 2 will verify that there is another TMU with a higher priority ID and will switch back to slave mode. The new TMU master (previously TMU slave 1) will resume normal network management. So for the TMU Slaves, special attention should be taken when granting their ID addresses and their location in the DTM system.

Simultaneous Masters in a DTM Network

If for some reason the master starts to receive a config or *ping* message (triggered by another TMU with a lower priority ID) then this means that the other TMU is not aware that there is already a TMU master. The TMU master with higher priority will try to notify, sending *ping* commands to the other TMU that it must assume slave mode. If the situation prevails after the TMU master sent 3 *ping* commands then the TMU master will raise an alarm and go to a fail mode. The TMU will not interact with the system when it is in fail mode, thus ensuring that faulty node is silent. Again, the number of *ping* commands sent by the TMU master in this situation is based on a theoretical assumption; 3 attempts are sufficient to assume and enter a fail mode. Sending only one *ping* command could be dangerous since this message could be corrupted and not reach its destination, therefore at least 2 *ping* commands should be sent to guarantee that there are no false assumptions due to message corruption.

4.5 NSU Management and Configuration

The communication between the TMU Master and each NSU is done via CAN messages. As mentioned before, the TMU master is the only entity that will initiate a communication in the DTM (excluding the normal traffic between nodes). This means that each NSU will only send a message upon a request from the TMU master. The data field of the CAN frame will contain the command and parameters. Each NSU must be plugged on to the system one at a time and in an orderly manner (NSU closest to the TMU Master until the last NSU which will be near the TMU Slave on the opposite extremity). Another precondition is that the NSU address must be configured to a default address before insertion into the system. A discovery and configuration command is sent by the TMU Master every 500ms to a predefined default address in order to verify and change the address of the new NSUs inserted in the system. The commands were defined for the communication between TMU and NSU and can be found in A2 - **TMU ARQUITECTURE**. The TMU master possesses tables with information about the location and address of each NSU.

NSU bring up and configuration

The following example demonstrates the NSU bring up and initial configuration.

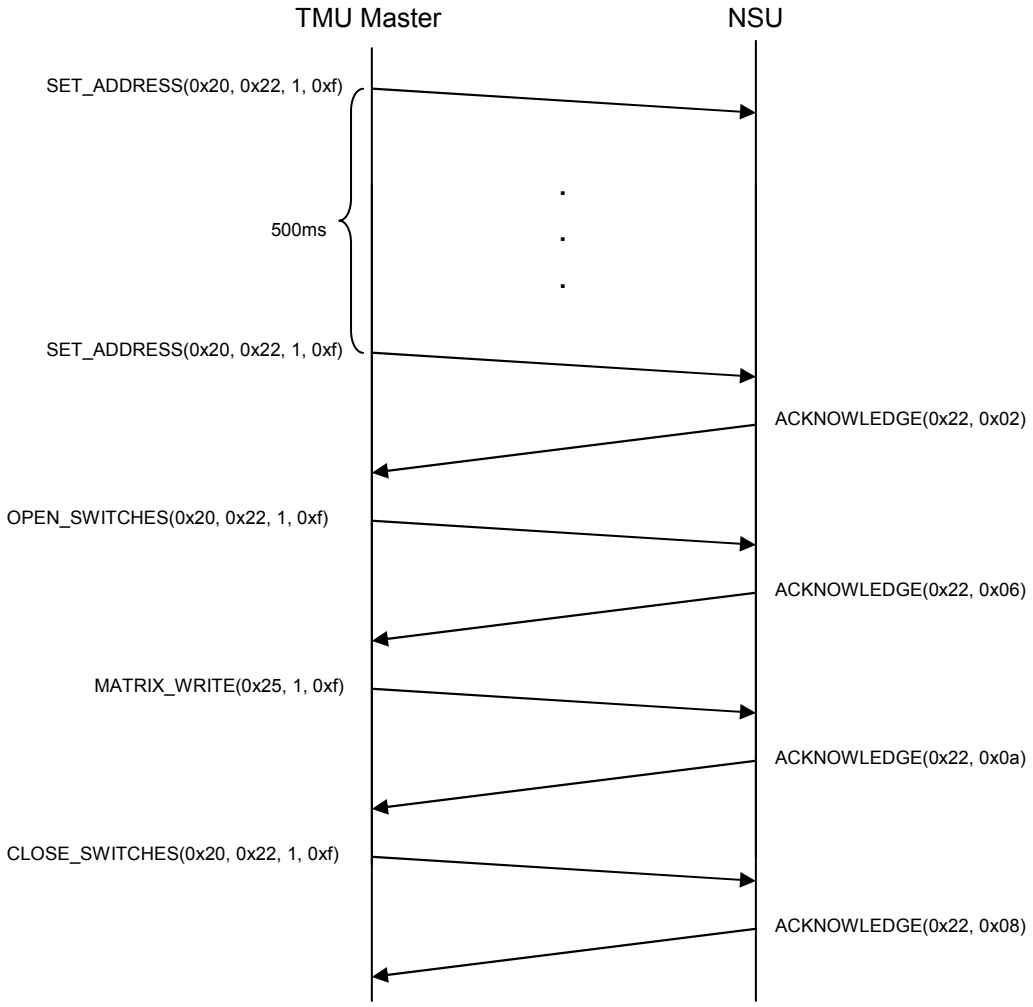

Figure 9 – NSU Bring up and Configuration

NSU normal operation

During operation, the TMU Master can change the switch table in a NSU as demonstrated in the following figure. The commands were defined for the communication between TMU and NSU and can be found in A2 - **TMU ARQUITECTURE**.

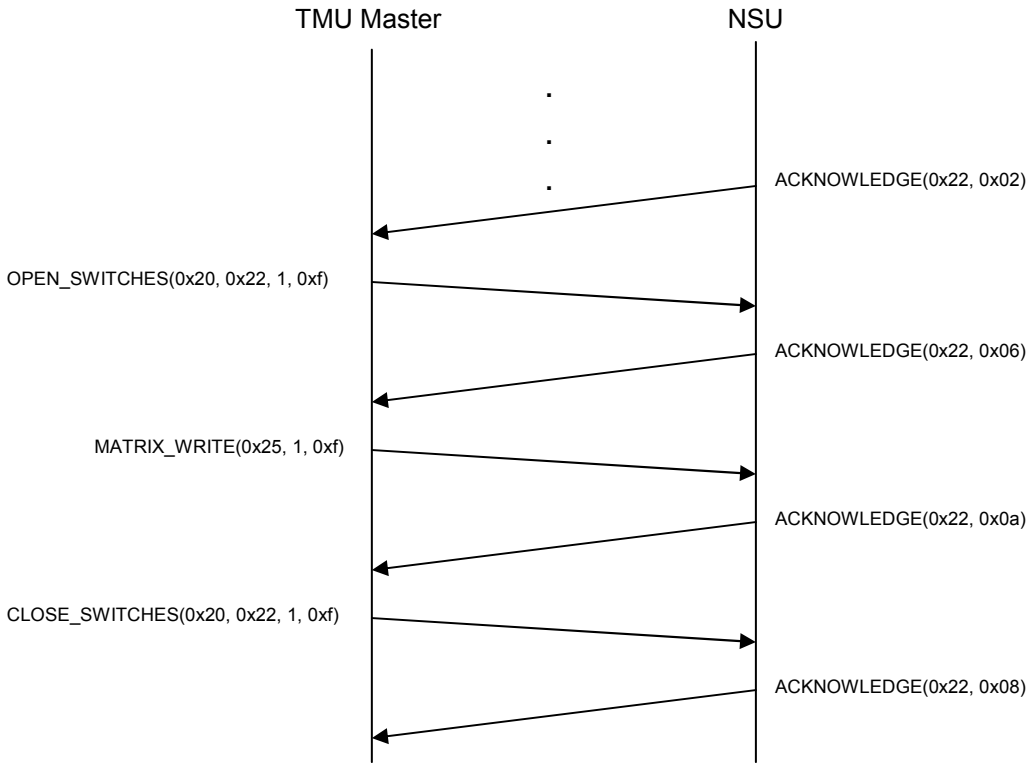

Figure 10 – NSU Normal Operation

4.5.1.1 Message Dispatch over the Media between TMU and NSUs

An acknowledge is always sent by the NSU upon the reception of a configuration or management message. This informs the TMU master of the success or failure of the dispatched configuration/management message. In case of failure the TMU master can quickly resend the message. Besides this acknowledge scheme, the TMU master dispatches the NSU configuration/management messages over more than one communication medium, thus improving the success rate and reducing the chances of resending the same message. To take the most advantage of redundant media and bandwidth efficiency, configuration/management messages will not be dispatched on all communication medium. Instead, the situation is similar to the synchronization message dispatch between TMUs. The configuration/management messages shall be dispatched on the same communication medium as the synchronization messages. This information is present in the NSU tables.

4.5.1.2 Bus Partition

To make use of the media, even though there is a partition, it is necessary to identify the location. Bus partition can be checked by sending *ping* messages to each NSU and TMU slave and verifying their response. The TMU master topology table has information related to the location of each element in the DTM network. This knowledge could help increase the efficiency of the TMU management. A bus partition can still be used by some of the nodes while other nodes make use of the healthy buses. This means that sometimes the DTM can partially use a bus which has been partitioned instead of totally isolating this bus. This has the advantage of using all available resources without waste.

However, partial use of a partitioned media may not be as straightforward as described above because of the lack of media termination. Other measures should be taken, like reduction of the baud rate transmission (TMUs and NSUs) in order to reduce reflections on the media. Rise and fall time of each transmitted bit should also be increased for the same reason. The change in the baud rate of a certain media (while maintaining 1Mbit/s baud rate on the other media) could pose some technical problems and increase the complexity of the system. This feature could be fully addressed in future research.

4.6 TMU Configuration and QoS

The TMU configuration is done via RS232 which is controlled by a dedicated software application called 'TMU Configurator and Monitor - TMUCAM', located on a PC. Only the TMU Master needs to be configured by the application since the TMU Slaves are configured by the TMU Master via the CAN interface. Each time any table (NSU, Reconfiguration, etc) is changed, the TMU Master will update the respective table in all the TMU Slaves.

The serial communication between the TMU master and the TMUCAM will have fixed communication settings and a set of commands for TMU master configuration and quality of service monitoring, as seen in appendix A2.3 – **Serial Communication Protocol**.

The TMU contains 3 banks of data and a register bank that can be configured externally by the TMUCAM (see appendix B0 – NSU, Reconfiguration and Topology Table). The purpose of these tables is to supply the TMU with context information related to all situations that involve CAN medium failure and efficiency management. These tables are located in a RAM memory and their locations are shown in Figure 50 – **NSU Table** and Figure 51 – **Reconfiguration and Topology Table**. The memory will have the NSU addresses located at the beginning of the memory (address 0). Each NSU address can be 4 bytes long (32 bits). Following the NSU addresses are the NSU tables for each bus failure. Each NSU table contains switch information for each NSU present in the DTM network. After the NSU tables comes the reconfiguration table. The reconfiguration table stores procedures for each possible combination of bus failures. Finally, located at the end of the memory, is the topology table. This table contains information relative to the location of each NSU in the DTM network. Furthermore, it should be considered the option of having location information of all TMU slaves in the topology table. There is also the possibility of putting the NSU addresses in the topology table instead of allocating them in the initial section of the memory.

The Configuration Register Bank (see appendix A2.4 - **ConfigRegBank**) is a set of registers that define and configure certain parameters as CAN Bus baud rate, number of NSU nodes, number of communication media, etc.

Figure 11 and 12 are an example of a possible TMU configurator and monitor software, named TMUCAM. This software is executed on a PC with an RS-232 connection to the TMU master. It is still in an embrionary phase; however, a brief description of future features and research shall be described below. The TMUCAM can be divided in two functionalities:

Configurator

It is important that there is an offline tool which enables the preparation and analysis of the DTM network prior to it's bring up. Therefore, the tool needs to have some sort of configurator which will build the NSU, Reconfiguration and Topology tables depending on various input parameters. A graphical interface will facilitate the work of dimensioning the network and its elements. As an example, there should be a possibility of drag and drop of NSUs or TMUs and also the ability to graphically connect the desired buses to an NSU. The tool could also include information related to each NSU bandwidth and timing requirement.

Monitor

Network monitoring is also an important feature of the TMUCAM. This tool should have the ability to analyze the systems performance, recalculate the tables and deploy these new tables to the TMU master. The ability to log the system's performance for statistical analyses is also a 'nice to have' feature.

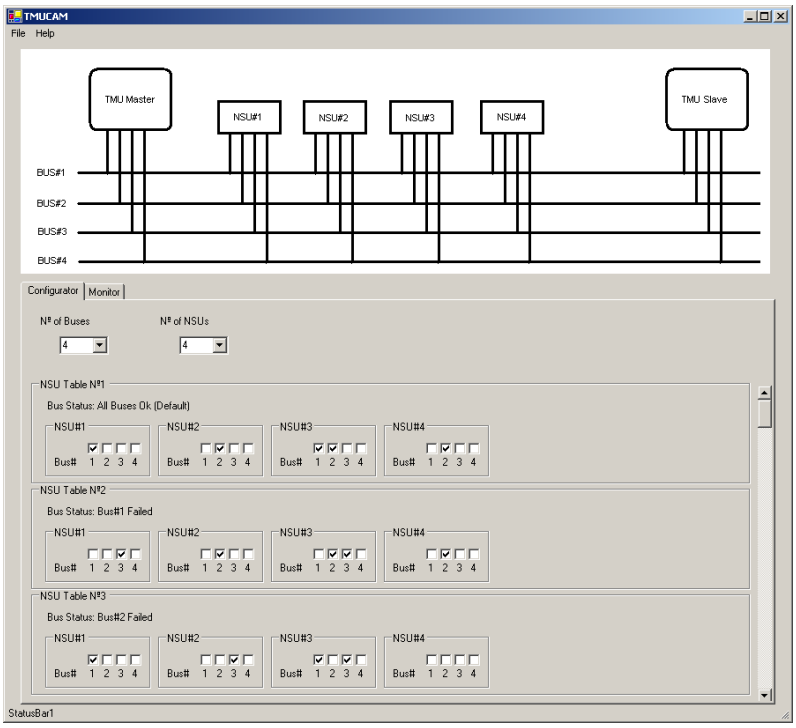

Figure 11 – TMUCAM Configuration Tab

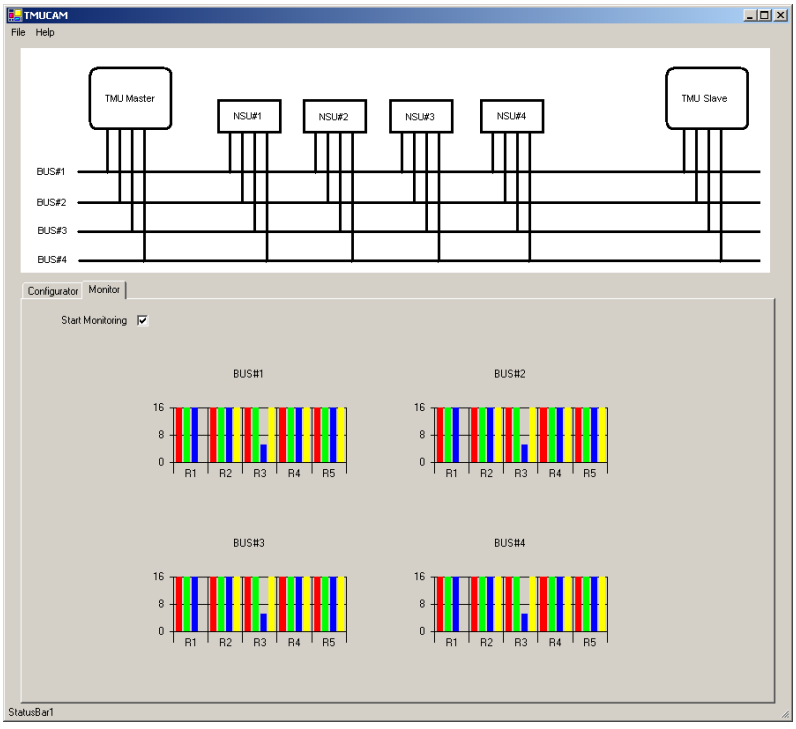

Figure 12 – TMUCAM Monitor Tab

The TMUCAM is out of the scope of this thesis and plenty of research on this software has to be done to take full advantage of the DTM network potential. It will play an important role in the systems performance and success.

4.7 TMU Brief Implementation Description

The TMU functionalities described in the previous sections can be implemented in blocks which communicate with each other via status and control signals. In Figure 13 – **TMU Internal Blocks**, a TMU internal architecture for FPGA implementation is proposed. Although the internal organization of the TMU is different from the one proposed by [3] and shown in Figure 3 – **TMU Internal Functionalities**, the behavior and objectives are the same. Most of the times the functional architecture and the implementation architecture diagrams are not coincidental, but the main ideas are fundamentally equivalent.

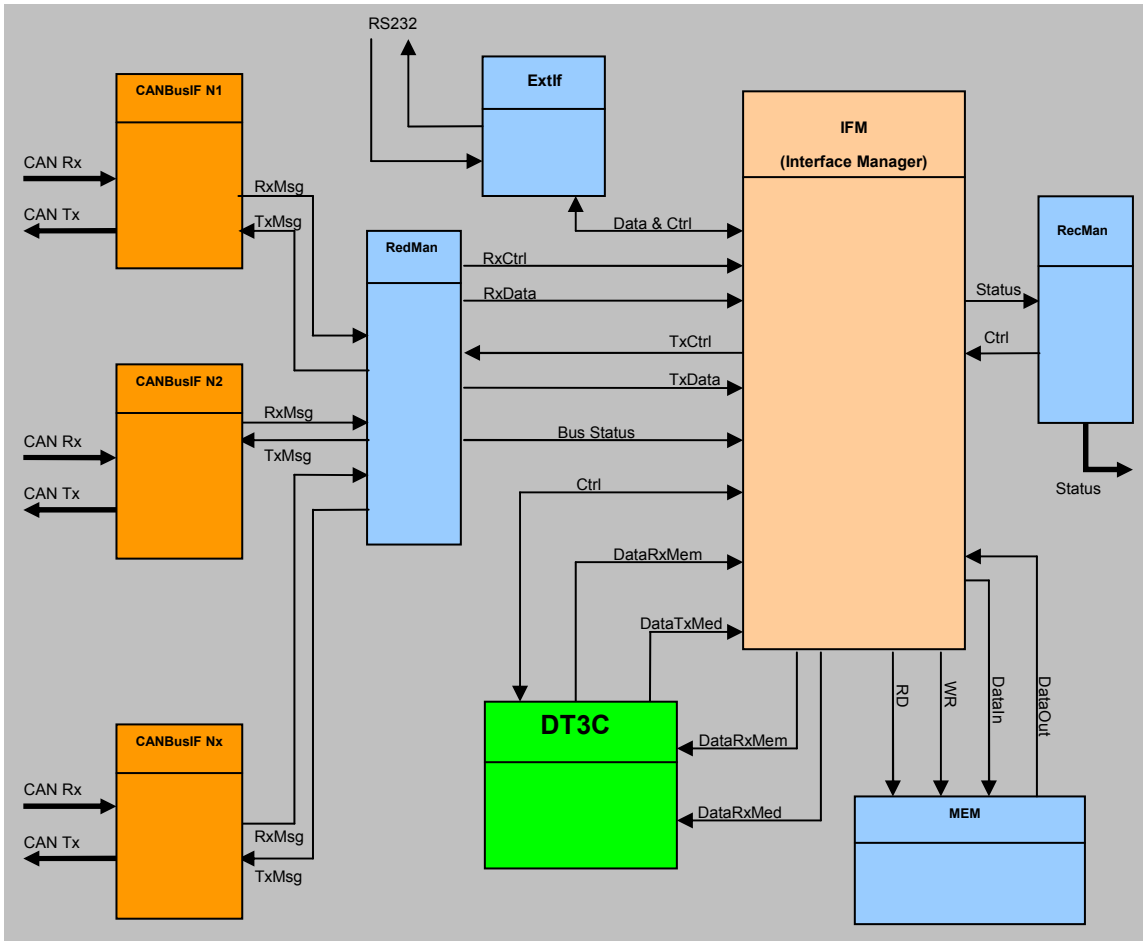

Figure 13 – TMU Internal Blocks

Besides identifying the best solutions/options for the TMU architecture, the main objective of this thesis will focus in a particular block inside the TMU; the Distributable Table Content Consistency Checker - DT3C. Each of the internal blocks of the TMU is briefly described below. The DT3C block will be described in detail and an implementation is also presented in the following chapters.

CANBUSIF

This module interfaces the CAN transceiver and is responsible for detecting faults on the transmission media (bfdUnit), as shown in Figure 14 – **CANBusIF Internal Blocks**. CANcore is an IP CAN 2.0B controller which interfaces the CAN bus transceivers. The bfdUnit detects failures and asserts it's only output signal in case there is a failure. The rxDataPath decodes each received message and identifies the type of incoming command.

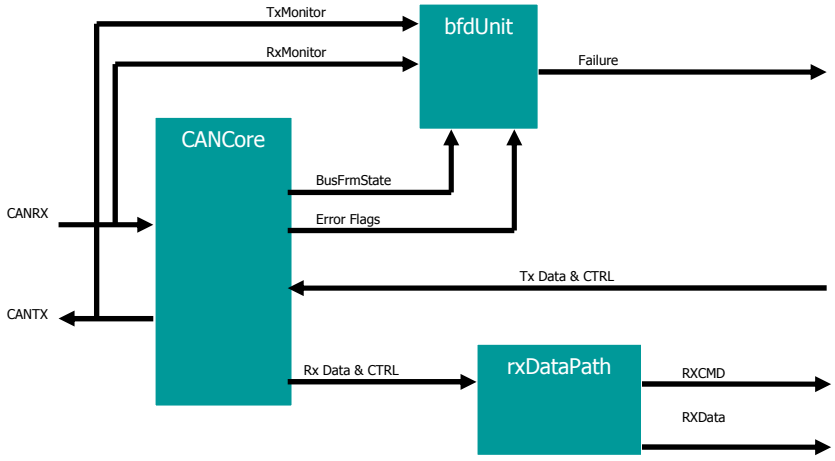

Figure 14 – CANBusIF Internal Blocks

RedMan

The block in Figure 15 – **RedMan Internal Blocks** is responsible for receiving and processing (cmpBus) the data from all CANBusIF. The cmpBus verifies that the data received from all CANBUSIF is the expected command. It then makes use of a majority vote scheme if there are any inconsistencies between the various commands. The txDataPath interfaces all transmit CANBUSIF. This block codes a command and its data to an adequate format for the CANBUSIF.

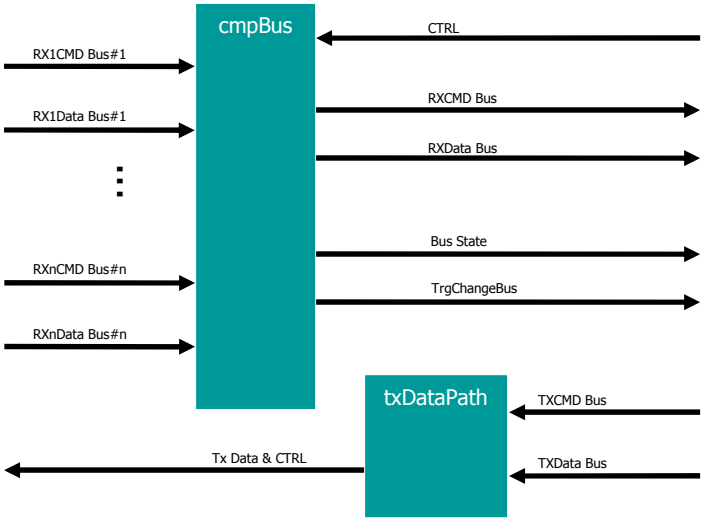

Figure 15 – RedMan Internal Blocks

EXTIF

The EXTIF block contains a uart and a small state machine that codes and decodes information from the PC (TMUCAM) and to the TMU master. The error flags from each CANBUSIF are directly connected to the Qos_TbReader, which permits a real time reading of the TMUs error status.

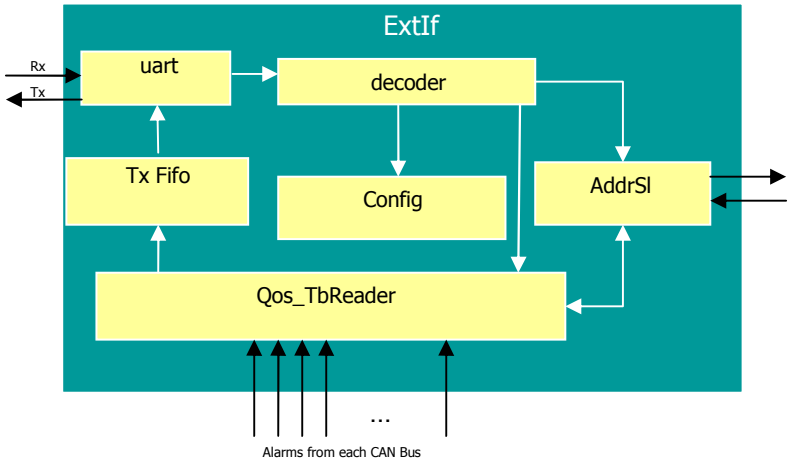

Figure 16 – EXTIF Internal Blocks

RecMan

The reconfiguration manager block determines the actual state of the DTM network and is responsible for dispatching the configuration commands for each NSU. The RecMan accesses the memory block to obtain information on procedures for actual network state.

IFM

The interface manager (IFM) is a memory access arbiter between the DT3C and the RecMan. Both blocks have read access to the memory, while only DT3C has both read and write access to the memory. The IFM is also responsible for filtering and dispatching each received command to its correct destination. For example, the content data (CRC info) of a received *ping response* is dispatched to the DT3C block.

MEM

The memory block is a RAM. This memory stores the NSU, Reconfiguration and Topology tables. It is accessed by the DT3C block and the RecMan block.

4.8 System Integration

The TMU is integrated in the DTM solution and there is a necessity to briefly describe the procedures to bring up the TMU and consequently the DTM network.

4.8.1 TMU Startup Procedure

At startup or hardware reset, the TMU is configured as slave and each CAN channel is configured at a baud rate of 1MBit/s. At this moment the DTM will only function at a predefined baud rate. This means that any element (TMU or NSU) that is plugged on to the system must be configured at the same baud rate (see appendix A2.4 – **ConfigRegBank**). The TMU Master has to communicate with the TMU Slave and interact with each NSU. In terms of internal functionality the TMU Master/Slave have the following behavior:

<table>
<tr><td colspan="2">1. After a startup or hardware reset the TMU is in slave mode by default and not configured. TMU awaits serial configuration (Master mode) or CAN configuration (Slave mode);</td></tr>
<tr><td>2. TMUCAM configures and sets the TMU as Master. TMU Master starts with the default NSU table (assumes all buses are functional);
3. After configuration, the TMU Master configures the TMU Slave via CAN (dispatches all tables to slaves);
4. After configuring the TMU Slave, the TMU Master executes 3 ping/ping responses to determine each bus state. If necessary, TMU Master sets the correct NSU table and notifies the TMU Slave.
5. Switch NSUReady LED on, indicating that the system is ready to accept NSUs.
6. TMU Master enters normal operation (At this time, each NSU can be attached or powered up in an orderly manner):
• Send Ping every 100ms;
• Wait and check for Ping response;
• After 4 Pings, send Set_Address. If there is an acknowledge from Set_Address, than the TMU Master sends configuration messages to the NSU.</td><td>2. TMU receives configuration data via CAN and enters Slave mode;
3. TMU Slave enters normal operation:
• Expect a Ping command every 100ms on at least one channel. Reply with a Ping response;
• If there is a lack of Ping commands for more than 300ms then switch to Master mode;
• If Config messages are received, then update the table and NSU state.</td></tr>
</table>

Table 7 – TMU Procedures

Various parameters mentioned above, such as the *Ping* (100ms) or Set_Address periodicity, are only theoretical values. These values must be adjusted and verified. Further research must be done on this issue to find the optimal values. The DTM performance and reaction times strongly depends on these values.

4.8.2 DTM System Bring Up

The example given only refers to two TMUs; one master and one slave:

1. Connect both TMUs at each end of the buses and in between connect each NSU to the buses;
2. Power up both TMUs;
3. Select one of the TMUs to be master by connecting a serial cable and uploading the configuration file created by the TMUCAM software;
4. Verify that the indicator TableSync is enabled in the TMU master and slave. This indicates that at present moment both TMU tables are synchronized.
5. As soon as the indicator NSUReady is enabled (in the TMU master), power up one NSU at a time, in order and only when the NSUReady is enabled. The established order is to start from the NSU closest to the TMU master, going along each NSU sequentially, until the last NSU which is closest to the TMU Slave. Each time a NSU is plugged or power up in the system, the NSUReady disables until the NSU configuration is completed;

After all NSUs are running, the system bring up is finished and ready for operation. The indicators mentioned in the steps above could be simple LEDs for visual inspection. Specific steps for the NSUs are not mentioned.

5 TMU Table Synchronization

Sharing a common set of data in a distributed system with a master and one or more slaves can have unexpected results if no precautions are taken to prevent data mismatch between these elements. Furthermore, in critical systems this data has to be consistent and available in these elements at all times, since any inconsistency on the replicated data can disrupt the system operation. This chapter proposes a specialized hardware architecture and a protocol, independent of the communication media, to prevent data mismatch between master and slaves for more than a predetermined time interval with a limited traffic overhead to the system. This hardware module is named Distributable Table Content Consistency Checker (DT3C). The DT3C module will be present in TMU master and TMU slaves.

This chapter begins by classifying the various types of distributed systems and describing the state of the art in the area of distribution of data. The description of the DT3C protocol is done in the subsequent subchapters.

5.1 Data Replication

Data replication is a common technique and plenty of research has been done in the areas of fault-tolerant services, distributed databases, distributed files, content delivery networks, distributed objects, and distributed shared memories. Different solutions for data replication and consistency management are proposed according to the constraints and objectives of each field. The problem between all these areas is that although they are conceptually similar, there is no common framework and therefore does not facilitate comparisons. An abstract and neutral framework for databases and distributed systems is provided in [23].

Replication in database systems is done mainly for performance reasons. The goal is to access data locally in order to improve response times and eliminate the overhead of communicating with remote sites. Fault tolerance is an issue but it is solved by using backup mechanisms which, being a form of replication, does not relate to the DTM system and therefore is out of the scope of this dissertation.

Distributed systems can be classified [23] as: synchronous or asynchronous. In a synchronous system a known bound is taken into account (i.e.: message transmission delay), while no such bound exists in the asynchronous model. The key difference is that the synchronous system allows correct crash detection, while the asynchronous system does not. There are four replication techniques in a distributed system:

- Active replication is a non-centralized replication technique. All replicas receive and process the same sequence of client requests. Consistency is guaranteed by assuming that, when provided with the same input in the same order, replicas will produce the same output. This assumption implies that servers process requests in a deterministic way.
- Passive replication occurs when clients send their requests to a primary, which executes the requests and sends update messages to the backups. The backups do not execute the invocation, but apply the changes produced by the invocation execution at the primary (i.e., updates). By doing this, no determinism constraint is necessary on the execution of invocations.
- Semi-active replication is an intermediate solution between active and passive replication. It does not require that replicas process service invocation in a deterministic manner. The protocol was originally proposed in a synchronous model.
- Semi-passive replication is a variant of passive replication which can be implemented in the asynchronous model without requiring any notion of views. The main advantage over passive replication is to allow for aggressive time-outs values and suspecting crashed processes without incurring a too high cost for incorrect failure suspicions.

This dissertation will focus in replicated data in a safety-critical distributed system with a Master/Slave topology in fieldbuses. In fieldbuses, previous work has been carried out in the area of data replication in order to avoid single points-of-failure and achieve fault-tolerance. A replication protocol [25] which makes use of the FTT-CAN [24] maintains consistent data between a master and its slaves. This protocol foresees data modification in the master during system operation and guarantees that all replicas are updated correctly and synchronized to the master by means of a special message called Trigger Message (TM). Nonetheless, this replication protocol does not worry about the bandwidth occupied during a synchronization procedure.

The DT3C is a synchronous passive replication protocol. A synchronous event is triggered at a constant time interval for the table verification between the master and slave. The lack of this synchronization query allows the slave to detect a malfunction on the master.

5.2 Data Distribution

The traditional packet format used in many protocols is shown in Figure 17 – **Traditional packet format and intermediate checksum packet format**. Most Automatic Repeat reQuest (ARQ) protocols [26] rely on checksums to let the receiver decide about the presence of transmission errors. If the checksum is wrong, the receiver provides the transmitter with appropriate feedback, which triggers a frame retransmission. When the channel bit error rate is not too high only a few bits are erroneous, but in a frame retransmission all bits are transmitted again, including the correct ones. In [27] the author has introduced the so-called intermediate checksum framing scheme (ICF), which attempts to rescue most of the correct bits and to restrict retransmissions only to those parts of a frame where bit errors actually occurred; a similar idea is briefly sketched in [27]. A distinguishing feature of the intermediate checksum approach is that it does not rely on coding, but requires only the ability to compute checksums. Many schemes have been devised to make effective use of the information contained in the erroneous packet copy. Such schemes are called type-II or type-III hybrid-ARQ schemes. A simple example of a type-II ARQ scheme is bit-by-bit majority voting: Once the receiver has received at least three erroneous versions of the same packet, it can guess what the received packet should be by applying a majority voting procedure to all bits.

Figure 17 – Traditional packet format and intermediate checksum packet format

Another scheme [29] addresses the problem of using a single checksum for the whole packet (as in the classical scheme). This cannot infer any information about positions of bit errors, so each bit is in suspect. By segmenting the packet into smaller chunks such that each chunk is equipped with an own checksum, the error information can be localized and the information in correct chunks does not need to be thrown away. A key advantage of this scheme is that it does not rely on coding but on (much easier) checksum computations. This allows implementation on top of COTS chip and furthermore checksum computations on the receiver side are much more energy-efficient than decoding algorithms. This idea is very similar to the concept used in the DT3C. The DT3C differs from [29] in the following ways:

- The checksum is calculated only over the data field of each received packet on the slave side. This means that for each received packet there is a corresponding checksum stored on the slave's local memory;
- A general checksum is done over all the checksums mentioned in the previous point. This general checksum is compared against the master's version.

- A tree algorithm is used on the stored checksums to find the corrupted chunk of information.

5.3 Fault Scenarios

The main purpose of the Distributable Table Content Consistency Checker (DT3C) is to avoid discrepancies of a set of shared data between the master and slave(s). Problems on the medium or on the node itself can cause various problems that can lead to data inconsistencies such as packet omissions, repetition or corruption. If the set of information is bigger than the data field of the packet, then subsets of information are created and orderly sent to the slave(s). The main issue is when the same subset of information is sent more than once (repetition) or not at all (omission). These problems can happen, for example in CAN, since there are known issues with multi-bit errors [30] and in the last-but-one bit of a CAN frame.

If the subsets of information are stacked in the slave regardless their relative ordering, then there will be a good chance of an information mismatch between the slave and its master. The DT3C solves the problem of packet repetition or omission by assigning and associating an identification number to each subset of information and calculating the Cyclic Redundancy Check (CRC) for the content (only a specific portion of the data field) of each incoming packet. The identification number indicates the address position of the associated data. This means that there is no indication prior to the transmission of the data of the total size or number of packets. Data is simply sent over the medium indicating the location in memory of its destination (no acknowledge scheme is used).

All CRC calculations are stored and used for comparison with the master, as it will be explained in more detail further on. The communication medium can also suffer disturbances that can corrupt the data content of the packet. Fortunately, most communication protocols use CRC [22][31] or some similar error detection mechanism. Although the CAN has a CRC calculation (from the Start of Frame until the data field), the DT3C applies the CRC calculation only to a specific part of the data field. Because this data is stored on a local memory in each node (master and slave) which can be accessed by various internal entities, the CRC calculation of the incoming data guarantees that data is not altered after reception. These CRC calculations are stored in a dedicated memory that only the DT3C has access. A general CRC calculation is done over all the previous CRC calculations and also stored in the dedicated memory. This general CRC is used to compare data consistency between master and slaves (this is explained in detail in section 4).

Although the DT3C is intended for the TMU, it was designed independently of the type of communication medium and therefore it will detect if there was data corrupted (only if the error occurs in a specific portion of the data field) during the transport of information between master and slave. This situation is less likely to occur in a CAN network since a packet with an inconsistent CRC is rejected by all nodes (slaves) and E. Tran [30] predicts an $1x10^{-7}$ error probability in a single CAN bus. However this situation is foreseen for communication mediums which have less robust error detection mechanisms such as the parity bit and also to foresee malicious (Byzantine) faults at the nodes capable of corrupting their internal memory or CRC malfunction. The issue of malicious faults is out of the scope of this work, but it will be addressed in future work.

We can summarize the faults that the DT3C proposes to solve:

1. Packet omission;
2. Packet repetition;
3. Packet corruption;
4. Packet unordered.

Cyclic Redundancy Check is a way of providing error control coding in order to protect data. Common CRC polynomials can detect the following types of errors:

- All single bit error;
- All double bit errors;
- All odd number of errors;
- Any burst error for which the burst length is less than the polynomial length;
- Most large burst errors.

Although this method is extensively used by the DT3C for consistency check between master and slave, the capability and efficiency of the CRC calculations will not be questioned. Furthermore, it will be assumed that the CRC polynomial can detect all errors previously stated.

5.4 DT3C Behavioral Description

Various terminologies and elements that are used throughout this chapter will be presented before proceeding to a behavioral description of the DT3C. A set of consecutive information that is shared between a master and a slave is defined as a table. Transmitting a table from the Master to the Slaves depends on factors such as the size of the table and the size of the packet data field. Therefore, a table is divided into several blocks of information (BoI) – Figure 18 – **General Packet Field Identification**. The size of each BoI depends on the size of the data field of the packet. Each BoI is identified with an Offset Information (OI). Most of the communication protocols are based on a similar packet structure, which can be divided in three fields; overhead, data and packet delimiter field, as shown in Figure 18 – **General Packet Field Identification**.

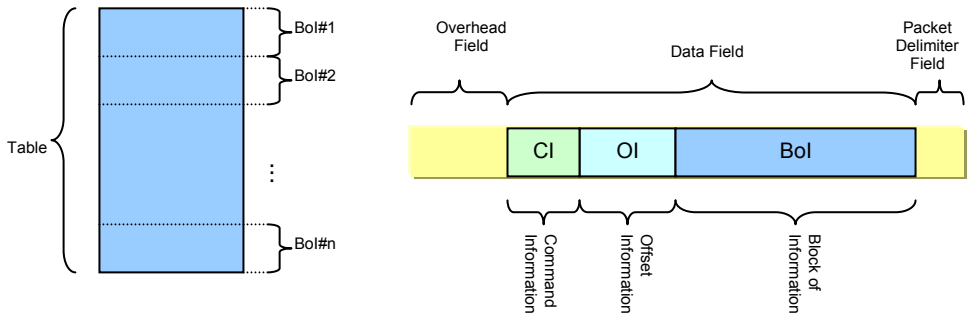

Figure 18 – General Packet Field Identification

The data field in a general packet structure can be divided into three sub fields:

1. Command Information (CI): this field indicates the type of information in the subsequent data field;
2. Offset Information (OI): indicates the location of the block of information.
3. Block of Information (BoI): contains a subset of the table content.

The DT3C module consists in five internal blocks and interfaces an external memory (such as a RAM). The external memory stores the table while the DT3C internal memory stores the various CRC data. The internal memory contents are initialized to zero at DT3C startup. The crcGen block is a 16 bit CRC generator. The dt3cFSM is the main block which controls all the other blocks. The datAddDec block manages the data and address bus. A detailed description of each block is made in appendix A3 - **DT3C Architecture**.

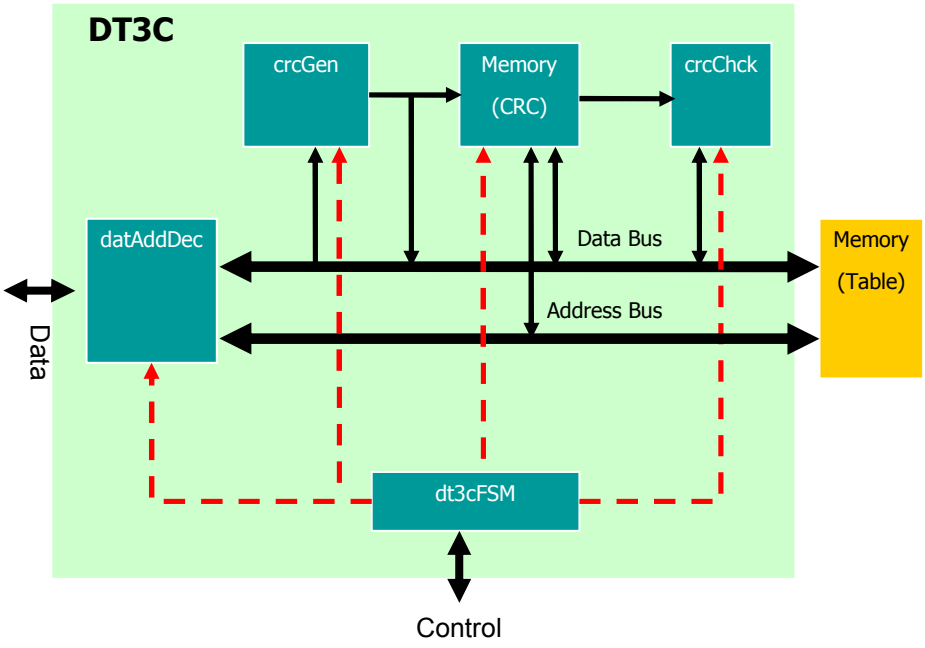

Figure 19 – DT3C Internal blocks

The DT3C can behave as a master or slave depending on its configuration. Only one master can be present in the system, while there can be more than one slave. The DT3C behavior (master or slave) can be categorized in four groups: Table Reception, Table Transmission, Table Verification and Table Synchronization. Each of these points is explained in the following subchapters.

5.4.1 Table Reception

The DT3C Master initially behaves as a slave in order to receive the table. This can be done by using a secondary communication medium to upload the table. For example, in the DTM system, the TMU master receives the table via uart port and distributes the table over the CAN medium to the TMU slaves. Each time the TMU master receives data over the uart port the internal DT3C module behaves as a slave. Because of the similarity of behavior between DT3C master and slave, the table reception will be described for the case where the slave receives the table.

In slave configuration, the DT3C starts off by waiting for the table to be sent from the master. At no circumstance will slave initiate any task or send messages without being ordered by the master. As each BoI is received, a BoI CRC calculation is done and stored in the CRC memory. Each time a BoI CRC is written to memory, the Table CRC is recalculated and stored in the CRC memory. The Table CRC is a CRC calculation of all data (BoI CRC's) present in the DT3C CRC memory. The Table CRC is stored in the internal memory at address 0. Suppose that a table has a dimension which can be divided into 5 BoI's. This means that the slave will receive each of the 5 BoI's. At the reception of each BoI the slave will execute the following steps:

1. Calculate the BoI CRC for the incoming BoI;
2. Store the BoI in the Table memory at the address indicated by the OI;
3. Store the BoI CRC in the CRC memory at the address indicated by the OI;
4. Read and feed into CRC calculator, all BoI CRC in CRC memory. Store the CRC result (which is the Table CRC) in the internal memory at address 0.

Figure 20 – **DT3C Memory** shows the correspondence between the internal memory (CRC memory) and the external memory (Table memory).

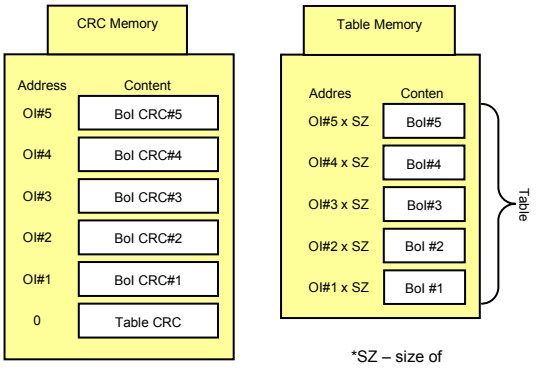

Figure 20 – DT3C Memory

If for some reason the slave receives the same packet twice, the last one (BoI) will prevail and the BoI CRC will be calculated and stored for this last packet. The Table CRC will also be recalculated for this last packet. If there is an omission of a packet during the transmission of the table then the memory block of this BoI in the external memory will contain garbage and its corresponding BoI CRC will not be calculated and stored.

5.4.2 Table Transmission

When the DT3C is configured as master and after receiving the table and building its own CRC memory, it initiates the table diffusion over the communication medium. This operation is only executed by the DT3C master. The master will read each BoI and verify its contents (calculating and comparing each BoI CRC) when dispatching the table over the medium. If the master detects that there is a mismatch, then an alarm is raised and all operations are ceased. This means that the table data in the master was corrupted after being dispatched by the management software - TMUCAM. Taking the previous example, the DT3C master will execute the following steps during table transmission:

1. Read BoI from Table memory to local buffer;
2. Feed BoI into CRC calculator;
3. Read the corresponding CRC BoI from CRC memory;
4. Compare the value obtained by the CRC calculator (step 2) with the CRC BoI (step 3);
5. Send BoI if previous comparison is equal, else raise an alarm and cease all operations.

The master will repeat the steps for each BoI until the entire table is sent over the communication medium. All BoI's are sent without any acknowledge.

5.4.3 Table Verification

After the table is sent to the slave, the master will periodically query the validity of the slave's table by sending its Table CRC (this information is embedded in the *ping/ping response* packet). The slave will receive these periodic messages which contain the Table CRC of the master. This value is compared against the Table CRC present in the slave. If this value is the same, then the slave simply replies to the master indicating an OK status. On the contrary, the slave will initiate the table synchronization process in order to obtain table consistency between master and slave. The following steps describe the table verification procedure:

1. DT3C Master: read Table CRC from CRC memory and send;
2. DT3C Slave: receive Table CRC;

3. DT3C Slave: read Table CRC from CRC memory;
4. DT3C Slave: compare master's Table CRC (step 2) with the slave's Table CRC (step 3);
5. Depending on the previous result, the table synchronization is triggered (Table CRC mismatch) or an "acknowledge" is sent to the master indicating the status success of the previous comparison.

The slave will always send a ping response independently of the result of the comparison. The content of the ping response, however, will contain the result of the table synchronization procedure in the case where there is a CRC mismatch between master and slave or it will simply contain an ok status (indicating no mismatch).

5.4.4 Table Synchronization

The table synchronization process is triggered by the slave each time there is a mismatch between its own Table CRC and the master's Table CRC in the table verification process. The table synchronization process consists in two procedures which are 1) CRC Memory Comparison 2) and Internal CRC Verification.

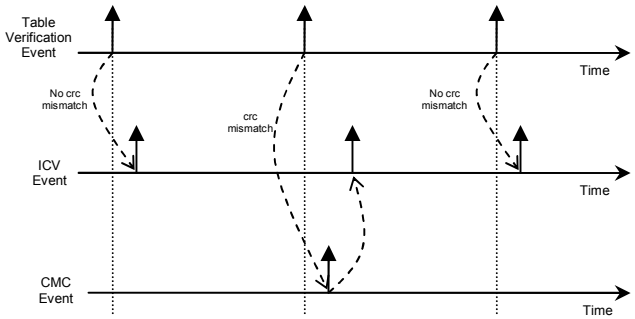

Figure 21 – Slave Synchronous Events

Both procedures have the purpose of finding a BoI mismatch between master and slave. The second procedure is executed only in the slave while the first procedure is executed in both master and slave.

CRC Memory Comparison - CMC

This procedure follows a similar approach to a tree search algorithm. A tree search algorithm seeks a set of sorted information by narrowing possible locations to progressively smaller intervals. In this case, the set of information are all the BoI CRC's present in the CRC memory (Table CRC is excluded). The strategy is to divide the CRC memory in two banks (excluding the Table CRC): lower and upper bank. Calculate the CRC of the lower bank of the CRC memory and send this value to the master. The master will also calculate the CRC for the lower bank of its own CRC memory and compare with the CRC received from the slave. If there is a mismatch then this means that the corrupted data is in the lower bank. This lower bank is again divided into two sub banks and the same procedure is done recursively until a single BoI CRC is reached. Figure 22 – **Corrupted Data Search Tree** demonstrates a possible path (dashed arrows) that could be taken to find the corrupted data in a CRC memory of size NCRCsz.

This procedure narrows down the corrupted data to a single BoI, in case that only one BoI has invalid data, otherwise the operation is repeated as many times as the number of BoIs that contain corrupted data. The default path chosen by the master or slave always begins by the lower bank of a certain level.

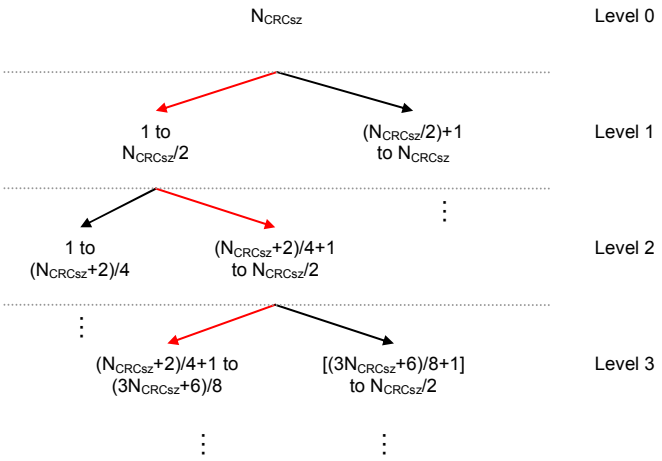

Figure 22 – Corrupted Data Search Tree

For example, in Figure 22 – **Corrupted Data Search Tree** the CRC memory is divided in two blocks – lower and upper bank level 1. The slave calculates and sends the CRC value of the lower bank level 1 to the master. The master confirms (by calculating its own CRC of its own lower bank level 1) that there is a mismatch of this value and divides this bank in two sub banks – lower and upper bank level 2. The master calculates and sends the CRC value of the lower bank level 2. The slave verifies that there is no CRC mismatch of the lower bank level 2. Therefore, the slave concludes that the corrupted data is in the upper bank level 2. Instead of calculating and sending the CRC value of the upper bank level 2, it will calculate and send the CRC value for the lower bank level 3. The procedure is repeated until a bank level n is equal to a single element, a BoI CRC. Level n can be reached on the master or slave side. If the level n is reached on the master's side, then the BoI is sent to the slave. If the level n is reached on the slave's side, then the slave will send the BoI CRC to the master which will confirm the CRC mismatch for the corresponding BoI and send it to the slave. The CMC algorithm is represented in figure 23.

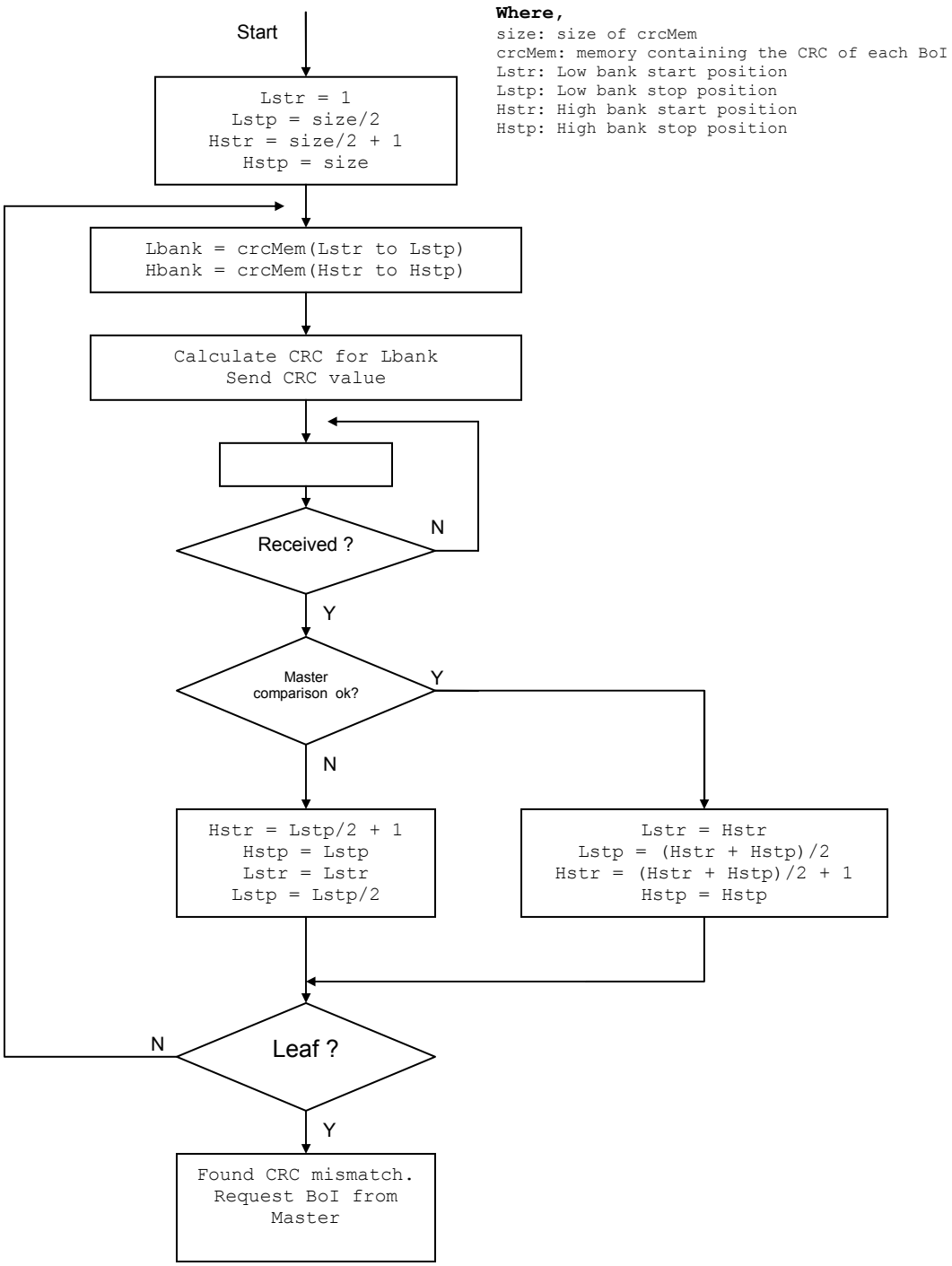

Figure 23 – CMC Algorithm

Internal CRC Verification - ICV

This procedure is executed first to verify if the CRC calculation done over each BoI in the Table memory corresponds to the BoI CRC located in the CRC memory. The slave reads each BoI from its Table memory, calculates the CRC of each BoI and compares this value with the BoI CRC stored in the CRC memory. At the end, the Table CRC is also recalculated. These CRCs are stored in the CRC memory, thus overwriting the previous CRC values. The steps executed for each BoI are:

1. Read BoI from Table memory;
2. Feed BoI into CRC calculator;
3. Read corresponding BoI CRC from CRC memory;
4. Compare value obtained by the CRC calculator (step 2) with the BoI CRC (step 3).

The ICV procedure guarantees that the BoI CRC's in the CRC memory corresponds to the CRC calculation of the BoI's in the Table memory. This procedure covers situations where there is any eventual tampering with the table memory (to which other devices/modules could have access). As shown in Figure 21 – **Slave Synchronous Events**, the ICV procedure is executed in the slave upon the Table Verification procedure or after a CRC mismatch to guarantee that any internal changes in the Table memory reflects in the CRC memory.

5.5 Dynamic Table Update

The growing demand for flexibility, mainly to improve the efficiency in using system resources requires that the DT3C is capable of updating the master's table and synchronizing its backup slaves in a short period of time. This operation is executed online and occupying the resources as less as possible. The DT3C master can have one or more BoI updated during system functionality. The master dispatches these BoIs over the communication medium indicating the OI. Each DT3C slave receives each BoI and stores it in the area designated by the OI. The CRC memory is refreshed each time a new BoI is received.

5.6 Multiple Slaves

The scalability of the DT3C depends on the number of TMU slaves that can be added into the system. The table transmission from master to slaves does not pose any problem, since this is a broadcast to all slaves present in the system. However, the synchronization uses a first come, first served approach. This means that when a table verification is executed, the master will receive from each slave one of the following messages:

1. an "acknowledge" indicating success, or
2. BoI request (slave ICV), or
3. CRC request (slave CMC).

All received messages are pushed into a FIFO in the master, which attends to each message one at a time. The reception of an "acknowledge" is registered by the master identifying the slave, and no action is required. With a BoI request, the master sends the requested BoI. A CRC request engages the CMC state machine in the master. During the execution of the CMC, the master will only attend to the messages of the slave which triggered the CMC. This allows the unsynchronized slave to regain table consistency as quick as possible. If various slaves send a CRC request, the first request is attended, while the rest of the requests are ignored. Only on the next table verification will one of the previously ignored requests have a chance in obtaining response.

In terms of scalability, the synchronization process does not impose any limitation on the maximum number of slaves and is very simple in terms of implementation, however not so efficient and some research can be spent on this topic to improve table synchronization of multiple slaves.

5.7 Communication Diagrams

At start up, the master sends the table through a number of packets indicating the block of information offset. As the slave receives the packets, it will calculate the CRC and store the BoI in its local memory indicated by the OI. As mentioned before, there is no acknowledge from the slave relative to the success or failure of each received packet.

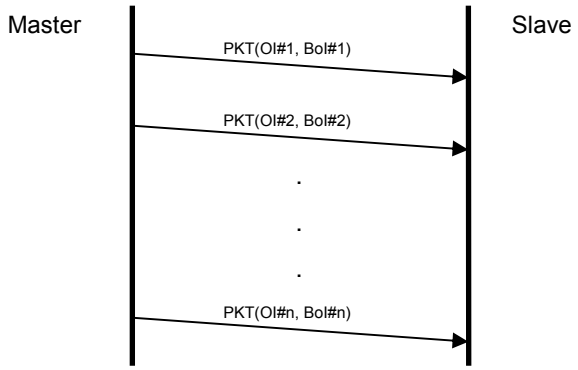

Figure 24 – Table Transmission/Reception

After a complete table transmission the master sends a periodic command (*ping / ping response*) embedded in the data field containing the Table CRC. Figure 25 – **Table Verification and** shows various possibilities of what can happen when the slave receives the master's Table CRC:

a) *Table verification:* table CRC match between master and slave; An acknowledge is sent to master confirming a table CRC match.

b) *Table verification and sinchronization (only internal BoI CRC verification procedure is executed):* table CRC mismatch between master and slave; and the slave identifies a mismatch in one of its BoI;

c) *Table verification and sinchronization (both procedures are executed):* table CRC mismatch between master and slave; and the slave cannot identify any mismatch in one of its BoI. Therefore the slave exchanges information with the master to narrow down the invalid data to a BoI. Master sends the updated BoI.

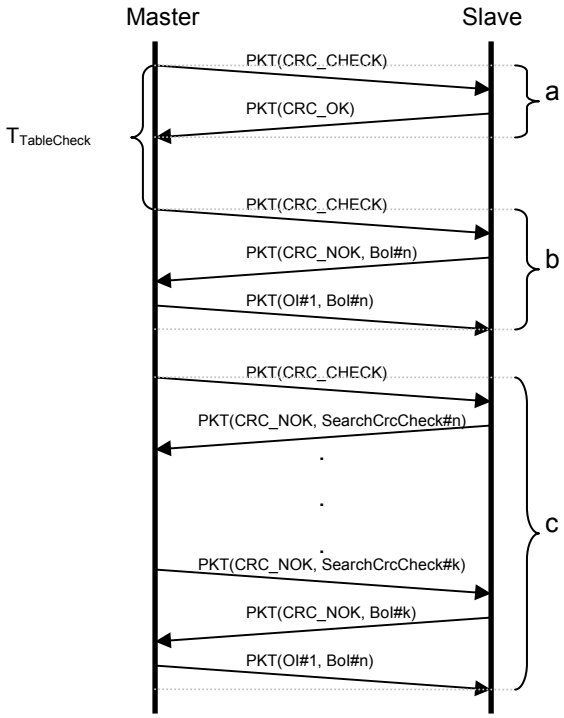

Figure 25 – Table Verification and Synchronization

5.8 Table Distribution Characteristics

5.8.1 Table Size and Packet Payload

There is a commitment between the sizes of the offset information, the block of information and the table. The number of packets (or number of BoI, N_{BoI}) necessary to send the table to the slave also depends on the three factors: OI, BoI and Table size:

$$N_{BoI} = N_{PKT} = \left\lceil \frac{T_{sz}}{BoI_{sz}} \right\rceil \quad \text{and}$$

$$N_{PKT} = 2^{OI_{sz}} \qquad (1), (2)$$

Where, N_{PKT} is the total number of packets necessary to transport the table, T_{sz} is the table size in bits, BoI_{sz} is the size (bits) of each block of information and OI_{sz} is the size (bits) of the offset information. In order to keep the communication medium available, the most important factor is the minimum number of packets needed to transmit the entire table. Therefore, it is important to find the minimum number of bits necessary for the offset information field and consequently the maximum size of the block of information that can fit in the packet. Combining equation (1) and (2),

$$OI_{sz_{MIN}} = \log_2 \left(\left\lceil \frac{T_{sz}}{BoI_{sz_{MAX}}} \right\rceil \right) \text{ (bits)} \qquad (3)$$

Equation 3 expresses the minimum number of packets to transmit a table and the compromise between the minimum necessary size of the offset information versus the maximum size of the block of information. Given a baud rate Bd_{Rate}, the packet size Pkt_{SzTb} and the number of packets N_{Boi}, the table transmission time is defined by:

$$BoI_{TxTime} = \frac{Pkt_{SzTb}}{Bd_{Rate}} \quad \text{and} \quad Table_{TxTime} = BoI_{TxTime} \times N_{Boi} \text{ (Seconds)} \qquad (4), (5)$$

It is also easy to verify that if the slave sent an acknowledge for each received BoI, the number of packets sent between both DT3Cs will be double. However the table transmission time may not increase to double because the acknowledge packet size Pkt_{SzAck} could be much less than a table packet:

$$Table_{TxTime} = \frac{\left(Pkt_{SzTb} + Pkt_{SzAck}\right) \times N_{Boi}}{Bd_{Rate}} \text{ (Seconds)} \quad (6)$$

Only if $Pkt_{SzAck} = Pkt_{SzTb}$ can be assumed that the table transmission time will double. Because the DT3C has no acknowledge scheme during the table transmission, the table transmission time is given by the following expression:

$$Table_{TxTime} = \frac{Pkt_{SzTb} \times N_{Boi}}{Bd_{Rate}} \text{ (Seconds)} \quad (7)$$

5.8.2 Table Synchronization Time

The table synchronization depends on the time the DT3C slave takes to execute a CMC and an ICV. In a best case scenario, the DT3C will find the only one corrupted BoI during the procedure, while executing CMC. But for the worst case scenario the CMC procedures would have to be executed many times to correct various corrupted BoIs. The maximum time for executing the ICV and CMC is important to determine the minimum time in which the master can query the slave for table CRC comparison. The following subchapters analyze all timings related to the table synchronization.

5.8.2.1 ICV Task Time

The ICV procedure repeats the following steps until it verifies all the CRC memory.

1. read a BoI from the table memory;
2. calculate the CRC;
3. compare the calculated CRC with the CRC stored in CRC memory;
4. if there is a mismatch, request BoI from master, otherwise do nothing.

It is considered that each BoI is n bytes and that the Table memory is 8 bits in width, then the tasks specified above can be done in parallel as shown in Figure 26 – **BoI CRC Verification Time**, for operations on one BoI.

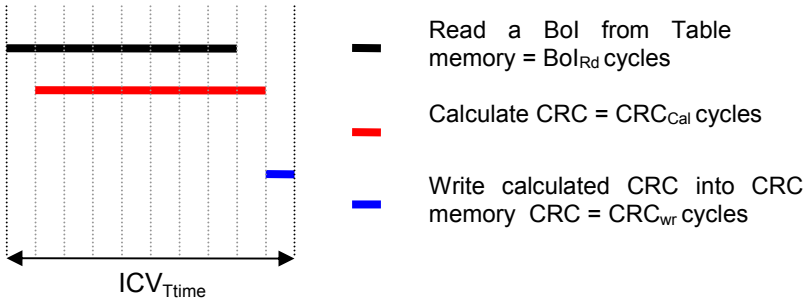

Figure 26 – BoI CRC Verification Time

So the ICV task time, ICV_{TTime} depends on the clock rate, Clk_{cycle} at which the DT3C is running and the number of cycles necessary to execute the steps specified:

$$ICV_{TTime} = Clk_{cycle} \times \left(N_{BoI} \times \left(BoI_{Rd} + CRC_{Cal} + CRC_{wr}\right)\right) \text{ (seconds)} \quad (8)$$

If the Table has a considerable size then reading a BoI from the Table memory while feeding this data to the CRC generator and writing the CRC to the CRC memory, basically equals the total time of these group of tasks to the task of reading the BoI from the Table memory. The previous statement is even more evident if the clock rate of the DT3C is much greater than the communication baud rate. Then we can consider that:

$$ICV_{TTime} = Clk_{cycle} \times N_{BoI} \times BoI_{Rd} \text{ (seconds)} \quad (9)$$

5.8.2.2 CMC Task Time

The CMC procedure is executed to find the corrupted BoI. As explained before, the CMC procedure uses a tree search algorithm to find the corrupted data. This means that there are two timings that are relevant in the CMC procedure:

- Time to calculate the CRC of an interval of CRC BoIs in the slave and master;
- Packet transmission time for slave query (T_{PKTQ}) and master response (T_{PKTR}) (*ping / ping response* command defined in A2.1 - **TMU Management messages**);

The first timing factor depends on the number of CRC BoIs stored in the CRC memory. The number of CRC BoIs needed on each CMC iteration decreases by half. For example, if there are N CRC BoIs in the CRC memory then on the first iteration of the CMC algorithm the CRC calculation is done on only N/2 CRC BoIs, on the second iteration it will be N/4. Reading a CRC BoI (Rd_{crcBoi} cycles)and feeding it to the CRC generator is done in parallel, therefore the total time for CRC calculation of CRC BoIs is:

$$CMC_{crcCal} = \left[\sum_{x=1}^{\frac{N_{BoI}}{2}} \frac{N_{BoI}}{2.x} \right] \times Clk_{cycle} \times Rd_{crcBoi} \text{ (seconds)} \qquad (10)$$

Where N_{BoI} is the number of BoIs (which is equal to the number of CRC BoIs in the CRC memory), Rd_{crcBoi} is the number of clock cycles necessary to read a CRC BoI into the CRC generator.

The second factor (T_{PKTQ} and T_{PKTR}) depends on the baud rate, the size of the packet and the number of packets sent and received during the CMC procedure. The packet size of the slave query is the same size as the master response packet. In order to determine the number of packets exchanged between the master and slave, the CMC tree search algorithm must be characterized. This tree algorithm is characterized in terms of the number of branches and its height (also designated levels, as shown in Figure 22 – **Corrupted Data Search Tree**).

$$Br = \left\lfloor \sum_{x=1}^{Height} 2^x \right\rfloor - 2 \times \left[2^{Height} - N_{BoI} \right] \qquad (11)$$

and

$$Height = \left\lceil \log_2 \left(N_{BoI} \right) \right\rceil \qquad (12)$$

The maximum number of exchanged packets to achieve the extremity of a branch (leaf) is given by the height of the tree. However if the master achieves the leaf, then it is necessary to exchange two more packets (slave request the specific BoI, and the master sends the BoI) to obtain synchronized data. The maximum number of packets exchanged, $N_{CMC\text{-}Packets}$, (worst case) to find and substitute a corrupted BoI is given by the following expression,

$$N_{CMC\_Packets} = \begin{cases} Height+2, HeightEven \\ Height+1, HeightOdd \end{cases} \qquad (13)$$

The total maximum time that the slave takes to synchronize with the master can be expressed as:

$$CMC_{Total_{Time}} = 2 \times CMC_{crcCal} + N_{CMC\_Packets} \times \frac{Pkt_{sz}}{Bd_{rate}} \qquad (14)$$

Notice that the CMC_{crcCal} is multiplied by two because this operation is executed on both sides, master and slave.

Again, if the clock rate of the DT3C is much greater than the communication baud rate then it can be considered that $CMC_{Totaltime}$ is approximately the packet exchange time from and to the master as,

$$CMC_{Total_{Time}} = N_{CMC\_Packets} \times \frac{Pkt_{sz}}{Bd_{rate}} \qquad (15)$$

This concept could be taken a step further using N x CMC algorithms in parallel in order to diminish the $CMC_{Totaltime}$. The CRC memory could be divided into N sections and each section would have its own CMC algorithm. The limitation here is the space required in the data field of the slave query and the master response packet (*ping / ping response* command defined in A2.1 - **TMU Management messages**) to carry simultaneously N x 16bit CRCs for each CMC algorithm. By dividing the CRC memory and applying the CMC algorithm only to each portion, the corrupted information can be found much faster since the height of each search tree is smaller. On the other hand, the packet size of the slave query and the master response will increase due to the fact that it has to carry one CRC value for each CMC algorithm. To take real advantage of parallel CMC algorithms, there must be some mechanism to realocate space in the data field for the CMC algorithms that have identified a mismatch. There is no point in occupying space in the data field with a CMC algorithm that has not identified a mismatch in its portion of the CRC memory. Therefore, this dedicated space in the data field could be reallocated to the CMC algorithms that have identified a mismatch (for an enhanced CMC version) or simply left out, thus decreasing the data field size.

6 DT3C Implementation and Results

6.1 Overview

Two approaches were taken in order to validate all the theory described in the previous chapter (5-TMU Table Synchronization). The first approach consisted in using Matlab simulation scripts to prove that it is possible to use CRC calculation in a recursive form by means of a search tree algorithm (6.5.2 - CMC Algorithm Simulation). The second approach consisted in describing the DT3C in a HDL (VHDL) and had the objective to prove that it was possible to implement the DT3C in hardware. A brief description is made on the HDL procedure, flow, tools and rules that were used throughout the implementation of the DT3C.

6.1.1 DT3C Design Considerations

The DT3C was implemented as a general and technology independent FPGA hardware module for table synchronization between the master and a slave. This module was conceived so that it can be used in a master/slave topology in fieldbuses, (CAN, FIP, PROFIBUS, modbus) or point to point communications such as RS232 or USB. This means that the DT3C module will have an interface which will be independent of the communication level.

The implementation of the DT3C in FPGA has the following characteristics:

1. Support for only one master and one slave;
2. One CMC algorithm can be executed at a time;
3. Technology independent;
4. Communication level independent;

The specification of the DT3C is done at a functional and RTL level and can be found in appendix A3 - DT3C Architecture.

6.1.2 HDL Design Rules

A consistent design methodology [21] was used throughout the DT3C design. These methodologies are correlated to issues such as clock and reset distribution and usage, shift registers, synchronization and asynchronous logic. The objective of these rules is to achieve a stable design, avoiding metastability problems, race conditions and glitches. The following rules are followed closely in the entire design:

1. Use only synchronous circuits;
2. Synchronize asynchronous events safely;
3. Use only one clock signal for a synchronous circuit (subsystem);
4. Use a master reset signal for circuit initialization during power-up;
5. Use D-type flip flops only;
6. In case of timing problems use pipelined circuits;
7. Avoid tri-states on chip due to reliability requirements;
8. Keep the design layout independent;
9. Circuit's function should not rely on a specific timing behavior.

6.1.3 Design Flow and Tools

The design follows a number of cyclic tasks which guarantee the success of the implementation and avoid common and simple errors. The first task and most important is to define and specify the module. The actual implementation can only start after the specification is stable and well defined. Attempts to make shortcuts in this first procedure can cause an erratic design and drastically increase implementation time. The DT3C is divided into sub modules and each of these sub modules are divided into components. The idea is to implement each component separately. A syntax and synthesis check is executed for each of these components. The component validation is finalized with a functional simulation. All components are integrated into each sub module and the previous tasks are executed: syntax, synthesis and functional simulation. After obtaining all valid sub modules, they are finally integrated to form the module. As shown in Figure 28 – **DT3C design flow**, again all tasks are executed and additional tasks (place and route and netlist download) finalize the design flow.

The following tools were used in this project to execute the tasks described above:

- Xilinx ISE 8.2 – used for vhdl syntax verification, synthesis, place & route and netlist download;
- Synplify – used for vhdl syntax verification and synthesis;
- FpgaAdvantage – used for vhdl syntax verification and for documentation;
- ModelSim – used in VHDL functional simulation;
- Tortoise CVS – file version control system.

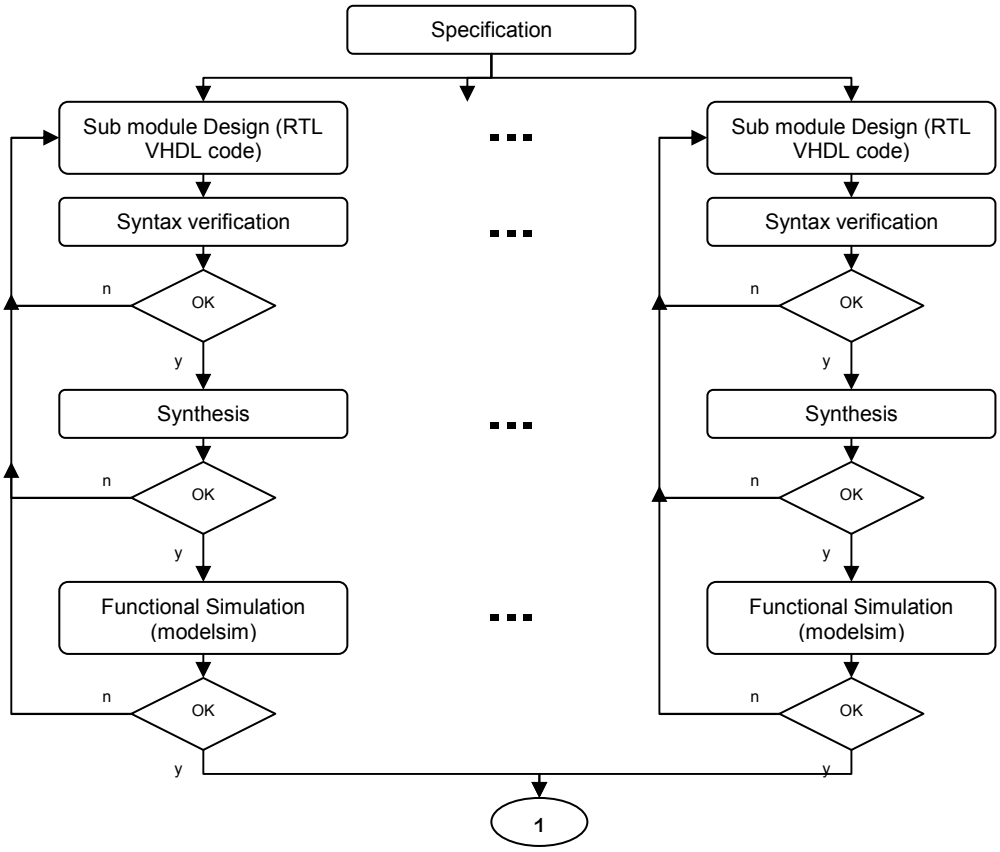

Figure 27 – DT3C design flow 1/2

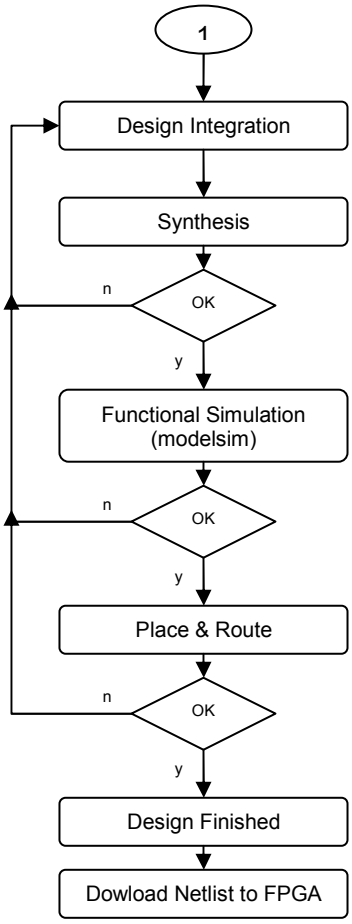

Figure 28 – DT3C design flow 2/2

6.1.4 Design Hierarchy

The DT3C specification in appendix A3 - **DT3C Architecture** is done based on the levels in Figure 29 – **DT3C Hierarchy**.

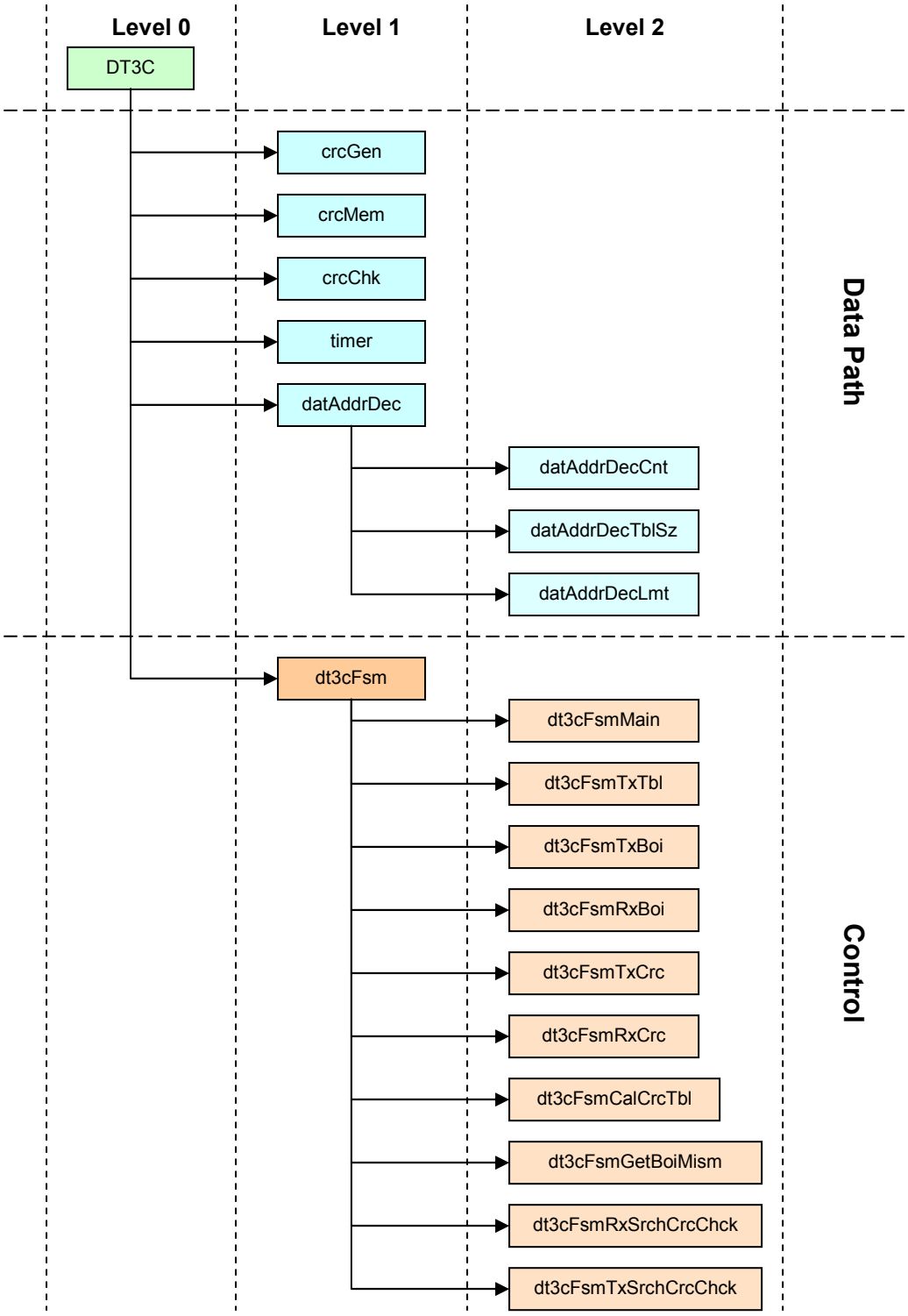

Figure 29 – DT3C Hierarchy

6.2 Clock Distribution

The DT3C module has only one clock domain. The master clock is an input of the DT3C module and internally distributes the clock signal to all entities as shown in Figure 30 – **Internal Clock Distribution**. Because there is only one clock domain, there is no need for clock domain data transfer logic between the entities. However if the DT3C is used as a component (i.e. TMU) in a multi clock domain, special care should be taken to interface the DT3C signals to avoid metastability.

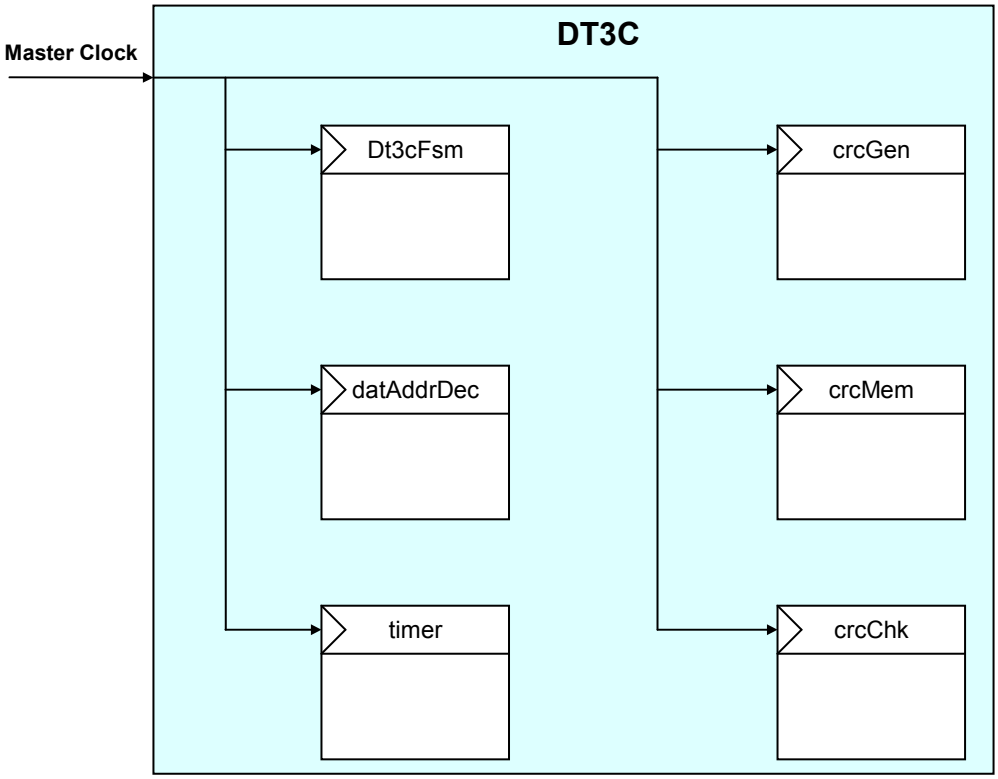

Figure 30 – Internal Clock Distribution

6.3 Simulation

6.3.1 Logic Simulation

6.3.1.1 Test Case crcChk module

This tests the functionality of the CRC comparator logic. The testbench consists in inputting various values on both the CRC data buses and verify the behavior when the setcrcReg is asserted. crcVld signal is asserted when both CRC data buses have the same content and the setcrcReg is active high. The crcVld signal will be asserted as long as setcrcClrReg is not activated. This means that before each comparison, the buffer must be cleared (setcrcClrReg) and only then the setcrcReg can be asserted. The following figure shows the correct behavior of the CRC comparator logic.

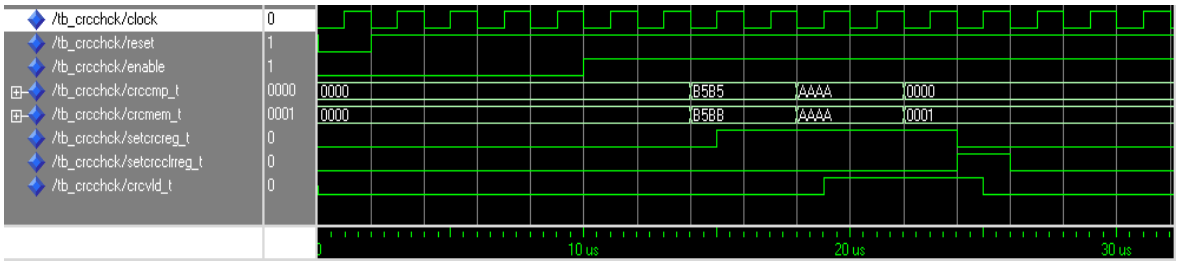

Figure 31 – crcChk Simulation

6.3.1.2 Test Case crcGen module

This tests the functionality of the cyclic redundancy check generator. The testbench consists in inputting two bytes 0xB5AA into the crcGen module and verifying that the result of the CRC calculation is 7C7D. The following figure shows that after inputting both bytes the crcReg is set to the desired value.

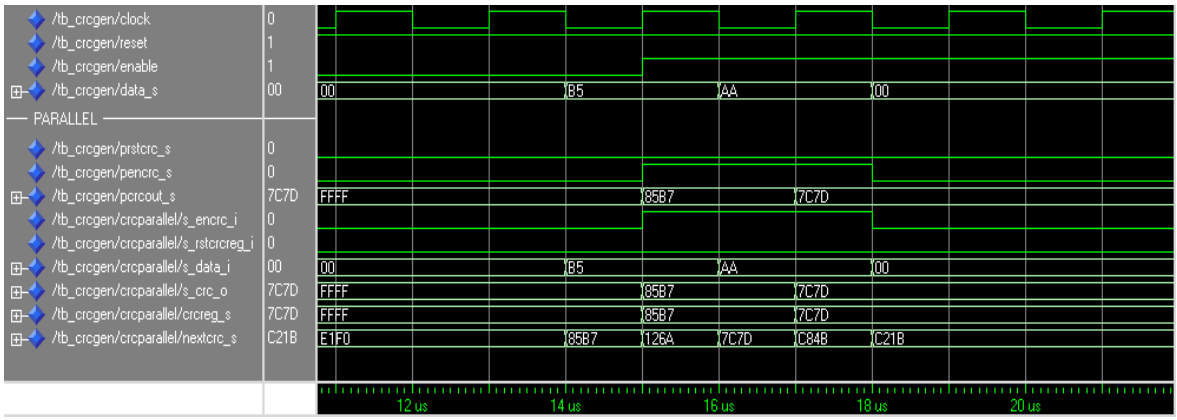

Figure 32 – crcGen Simulation

6.3.1.3 Test Case crcMem module

This tests the functionality of the CRC memory. The testbench consists in writing data into the CRC memory and reading the same values back. The following figure shows how all data read from the memory at the addresses 0 to 4 corresponds to the same values and addresses previously written to the memory.

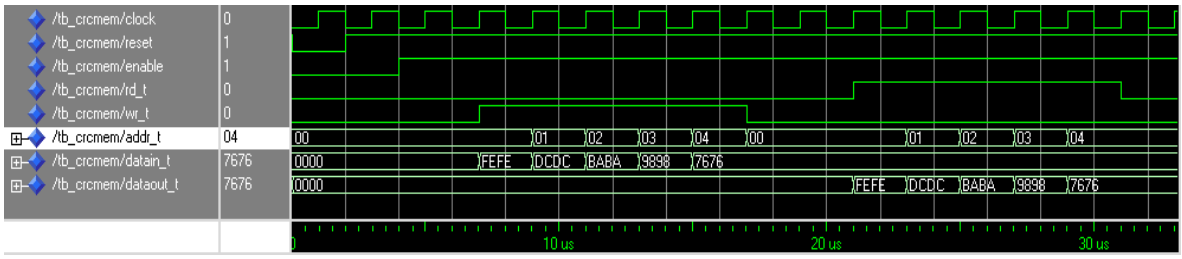

Figure 33 – crcMem Simulation

6.3.1.4 Test Case timer module

This tests the functionality of the timer module. The testbench consists in toggling the reset signal and verify that the timer does not trigger. The timer's internal counter value was altered for this test. The following figure shows that the timer triggers after an internal counter reaches 3. At any moment the timer can be reset and its internal counter is set to zero.

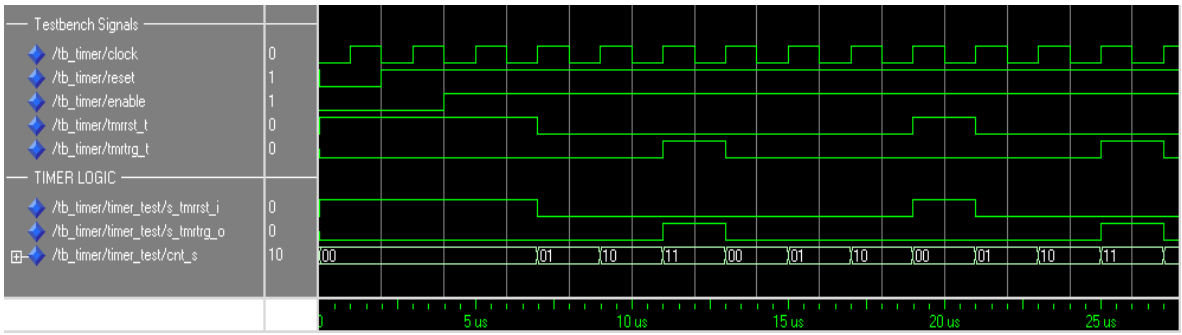

Figure 34 – Timer Simulation

6.3.1.5 Test Case datAddrDec / datAddrDecCnt module

This tests the functionality of the datAddrDecCnt. The testbench consists in verifying the start (strCnt) and stop (stpCnt) limits of the counter. Reset and counter increment are also tested as shown in the figure below.

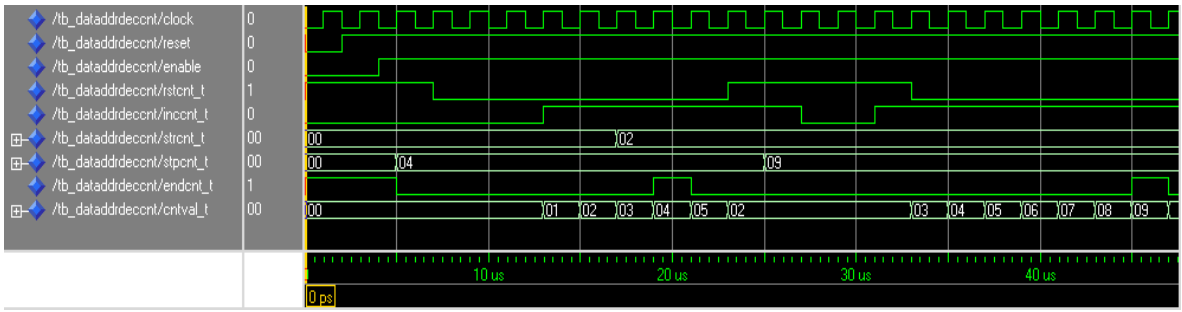

Figure 35 – datAddrDecCnt Simulation

6.3.1.6 Test Case datAddrDec / datAddrDecLmt module

This tests the functionality of the datAddrDecLmt. The testbench consists in verifying the ability of the design to store values (address range). The upper limit register is set to 0x04. The reset signal is asserted and the upper and lower registers are cleared. The upper limit register is set to 0x08 and a test is done to verify if this limit is lowered until 0x02. These test procedures can be seen in the figure below.

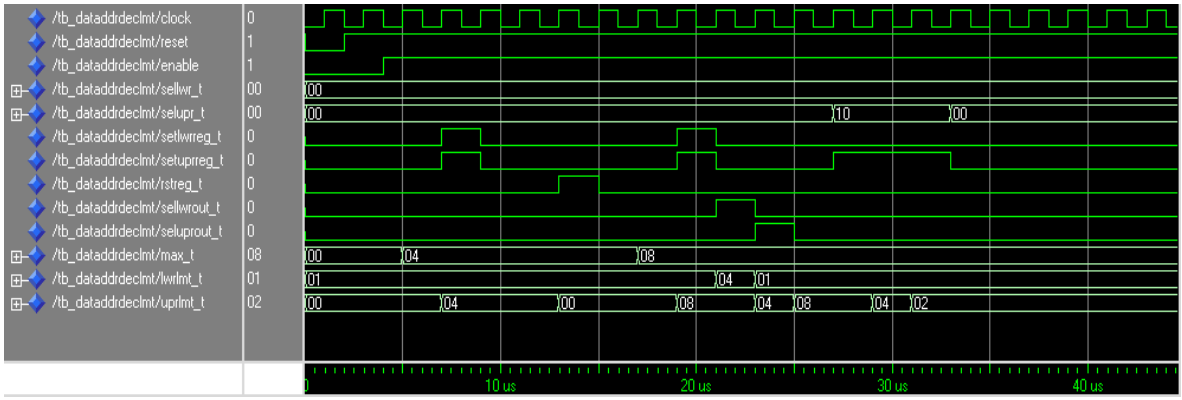

Figure 36 – datAddrDecLmt Simulation

6.3.1.7 Test Case datAddrDec / datAddrDecTblSz module

This tests the functionality of the datAddrDecTblSz. The testbench consists in two stages: incrementing the address bus until 0x07 and then decreasing the value until 0x00. In the first stage, the output (max_t) will be updated for every address increment. In the second stage, the address bus is decreased but the value on the output must maintain the previous maximum value (0x07). This test (with both stages) can be seen in the figure below.

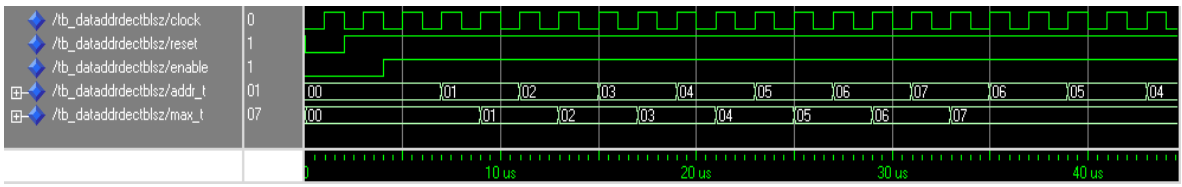

Figure 37 – datAddrDecTblSz Simulation

6.3.1.8 Test Case datAddrDec module

This tests the functionality of the datAddrDec. This entity uses the datAddrDecCnt, datAddrDecLmt and datAddrDecTblSz components. These components were tested previously and therefore, only the functionality of shift register logic will be tested in this testbench. The testbench consists in two stages: Receiving data (0x01AABB) from the media channel and transmitting data (0xEEFF) from the media channel. This test (with both stages) can be seen in the figure below.

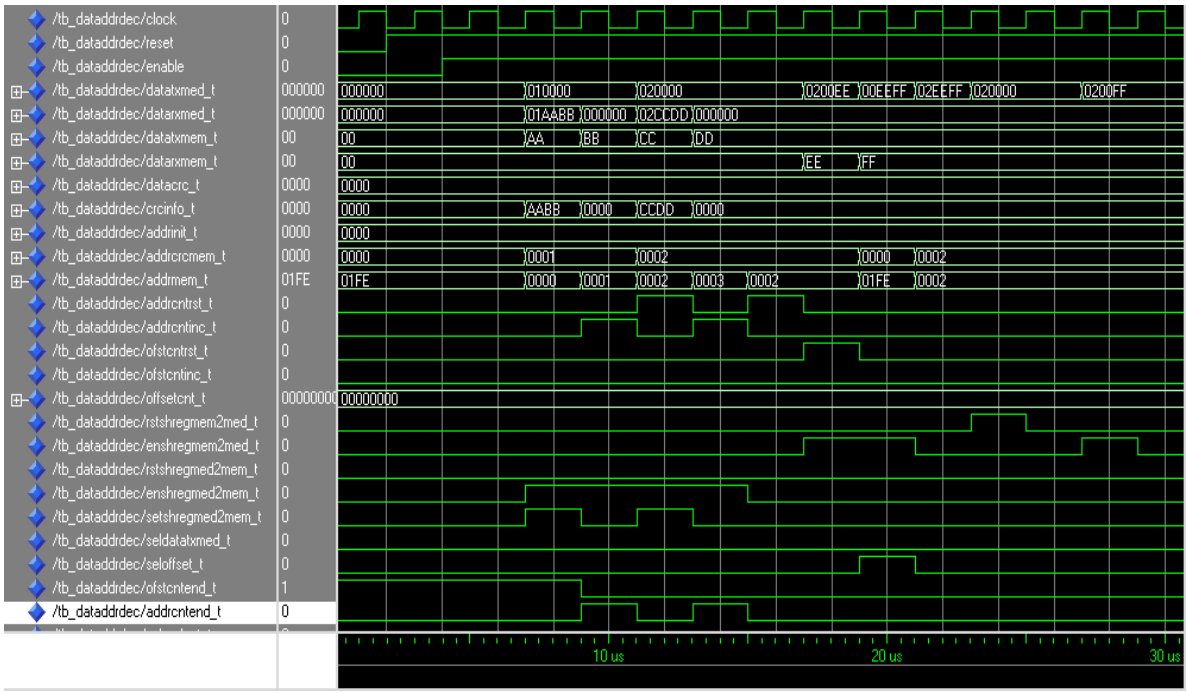

Figure 38 – datAddrDec Rx from Media Simulation

6.3.2 DT3C Module Simulation

This simulation has the intent of verifying the correct behavior of all state machines and logic in the DT3C module. The simulation is prepared by connecting a DT3C in master to a DT3C in slave configuration as shown in appendix A4 - **DT3C System Testbench** . This testbench could also be done using a communication media (i.e. CAN) between the DT3C master and slave. Additional glue logic is needed to connect the DT3C master signals to the DT3C slave signals. The following group of tests were executed:

1. Normal Operation:
   a) Input data to DT3C master;
   b) Verify the correct data dispatched to DT3C slave;
   c) Verify that master polls slave at a regular interval (CRC table check).
2. Failure Situation n1
   a) Input data to DT3C master;
   b) Stimulate a packet repetition during the table dispatch between master and slave;
   c) Verify that master polls slave at a regular interval (CRC table check), and no Table CRC mismatch has occurred on the first poll.
3. Failure Situation n2
   a) Input data to DT3C master;
   b) Stimulate a packet omission during the table dispatch between master and slave;
   c) Verify that when the master polls slave (CRC table check), the slave initiates a CMC and requests the missing BoI from slave;
   d) After initial conflict, verify normal behavior between both DT3C.
4. Failure Situation n3
   a) Input data to DT3C master;
   b) Stimulate a packet corruption during the table dispatch between master and slave;
   c) Verify that when the master polls slave (CRC table check), the slave initiates a CMC and ICV;
   d) Check the CMC exchange CRC data between slave and master. Verify if corrupted BoI was successfully updated;
   e) After initial conflict, verify normal behavior between both DT3C.
5. Update a BoI
   a) Input data to DT3C master;
   b) Verify the correct data dispatched to DT3C slave;
   c) Verify that master polls slave at a regular interval (CRC table check);
   d) After a period of time, update a BoI in the master and verify that this information is also updated in slave.

6.4 Synthesis

The DT3C was synthesized with Xilinx ISE and Quartus.

Altera	Xilinx
Selected Device: Cyclone2	Selected Device : Spartan2 2s30pq208-6
Estimated Total logic elements: 652 Total combinational functions: 652 Logic element usage by number of LUT inputs -- 4 input functions: 408 -- 3 input functions: 117 -- <=2 input functions: 127 -- Combinational cells for routing: 0 Logic elements by mode -- normal mode: 573 -- arithmetic mode: 79 Total registers: 450 I/O pins: 117 Maximum fan-out node: t_clk_i Maximum fan-out: 450 Total fan-out: 3849 Average fan-out: 3.16	Number of Slices: 274 out of 432 - 63% Number of Slice Flip-Flops: 199 out of 864 - 23% Number of 4 input LUTs: 513 out of 864 - 59% Number used as logic: 497 Number used as RAMs: 16 Number of IOs: 117 Number of bonded IOBs: 117 out of 132 - 88% Number of GCLKs: 1 out of 4 25%

Figure 39 – Synthesis Comparison

The number of logic elements between the Altera and Xilinx synthesis is different. The reason behind this difference is that the Xilinx Synthesizer is smarter and identified the CRC memory module as a RAM. The Quartus did not allocate the CRC memory logic as a RAM and therefore had to use extra logic (LUTs & logic). The other resources (dedicated clocks, IOs) are equal between both synthesis tools.

6.5 Table Dispatch and Synchronization Results

The traditional form of dispatching information to another node normally involves an acknowledge to determine the status of the received information. This method will be compared with the DT3C. The table synchronization has also a different approach in the DT3C. The DT3C synchronization procedure will be compared with the alternative of resending the entire table each time a resynchronization is requested.

6.5.1 Packet Size

As explained before, the number of packets necessary to send the table depends on equation 1, 2 and 3. In the CAN network, each data frame can contain a maximum of 8 bytes. If for example we use a config packet (2 bytes for command and identification information), then we will have 6 bytes for the OI and BoI. The figure below demonstrates the relation between the size allocated for OI and BoI (for a table which has 10240 bytes).

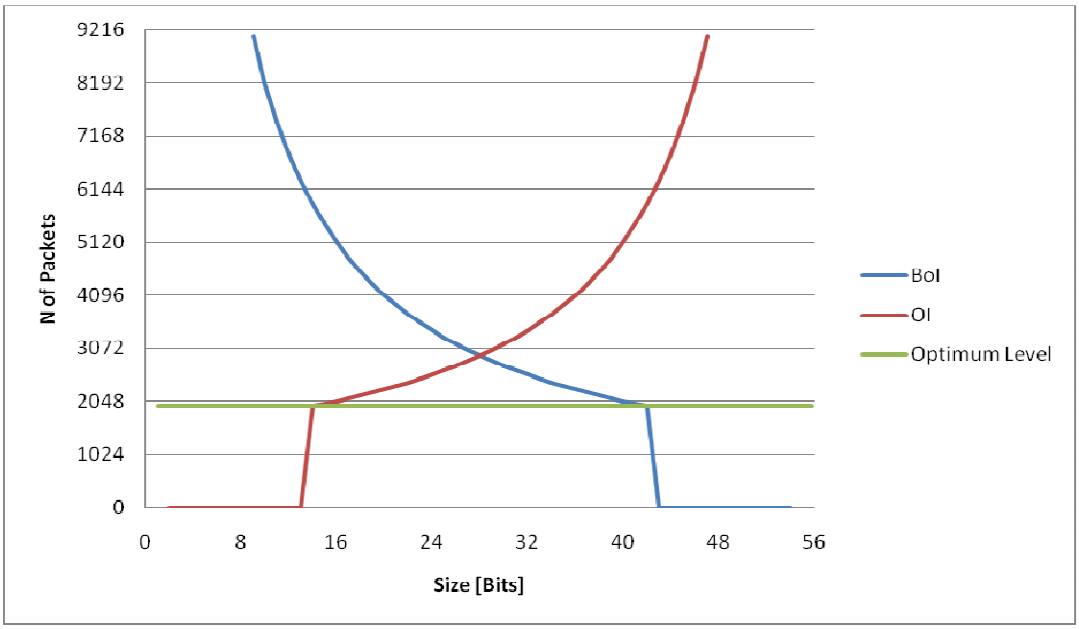

Figure 40 – BoI Size Vs OI Size for a Table = 10240 Bytes

The number of packets rises exponentially as the number of bits of the offset information increases. This also means that the number of bits of each BoI in a packet decreases as the offset information increases. The green line in Figure 40 – **BoI Size Vs OI Size for a Table = 10240 Bytes** indicates the minimum limit for the OI size. If this condition fails to be complied then there will not be BoIs with unique OIs.

6.5.2 CMC Algorithm Simulation in Matlab

A simulation of the CMC search tree algorithm was executed in matlab (algorithm can be found in Apendix A5 - **CMC Algorithm**). This set of matlab scripts served as a proof of concept in terms of recursive use of CRC calculations on data. The scripts consist in various modules:

- crc_range.m: executes CRC calculations on the input data and returns the CRC value.
- b_tmem.m: builds the Table memory for the Master and Slave.
- b_crcmem.m: builds two CRC memories, one for the Master and the other for the Slave.
- build.m: executes b_tmem.m and b_crcmem.m.
- cmc_algorithm.m: executes build.m and applies the CMC algorithm to the created CRC memories.

The purpose was also to confirm the minimum and maximum number of interactions to reach the leaf, therefore validating the equations in chapter 5.8. The horizontal axis is the number of BoIs, while the vertical axis indicates the number of iterations that the CMC algorithm takes to reach a leaf.

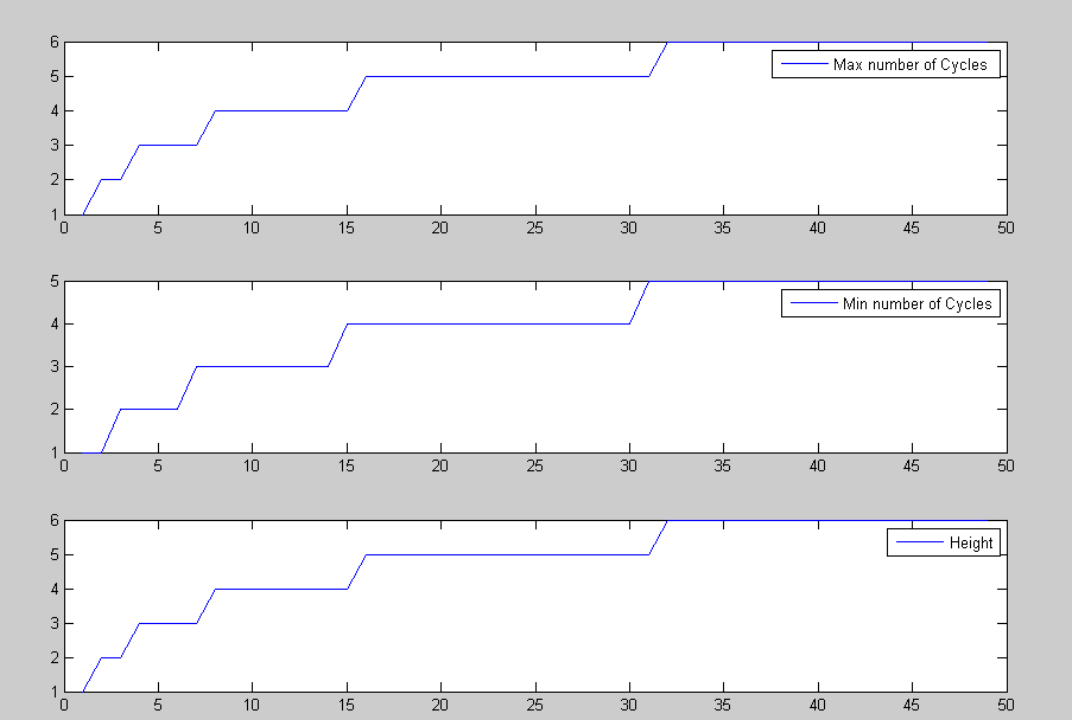

Figure 41 – Number of Cycles in CMC Algorithm

Depending on the number of BoIs, a tree has a maximum (worst case scenario) and minimum number of iterations to reach a leaf based on equations 11, 12 and 13.

6.5.3 DT3C Time Issues

The benefits of the DT3C from a perspective of bandwidth occupation will be demonstrated with the following environment:

CAN baud rate = 1Mbit/s

Table Size = 10240 Bytes

Standard CAN frame (ignoring stuff bits) = header + data + delimiter = 44bits + data

Where,

header = 19 bits

delimiter = 25 bits

and the data field depends on the type of command (defined in A2.1 - **TMU Management messages**):

Type of command	Data content	Data [bits]	Total Packet size [bits]
Ping	1byte [command] + 1byte [mbview] + 1byte [crcSt] + 2bytes [crc]	40	84
Ping Response	1byte [command] + 1byte [mbview] + 1byte [crcSt] + 2bytes [crc]	40	84
Config	1byte [command] + 1byte [type] + 2bytes [OI] + 4bytes [BoI]	64	108
Acknowledge	1byte [command] + 2bytes [NTadd] + 1byte [CCore]	32	76

Table 8 – Packet Size vs Command Type

Table Dispatch

The table is sent from the master to its slaves with config commands (A2.1 - **TMU Management messages**) and in a number of packets defined by equation 1, 2 and 3:

$$N\_Packets = \frac{10240}{4} = 2560 \text{ (Packets)} \qquad (16)$$

Notice that it is necessary that the OI occupies 2 bytes so that each BoI has an unique offset information. If there was an acknowledge mechanism for each received packet, then the total number of packets would double. However the total bandwidth occupation would not double because the acknowledge packet is smaller than a config packet. If we neglect the inter frame space between the packets and stuff bits, the total time that the communication medium is occupied for table dispatching, is:

$$TableDispatchTime_{with\_Acknowledge} = \frac{2560 \times (108 + 76)}{1Mega} = 471 \text{ (ms)} \qquad (17)$$

The DT3C does not make use of any acknowledge mechanism, so the total time is much shorter:

$$TableDispatchTime_{DT3C} = \frac{2560 \times (108)}{1Mega} = 277 \text{ (ms)} \qquad (18)$$

The DT3C occupies 41.2% less the communication medium than the solution which involves an acknowledge for each packet.

Table Synchronization

If there is a CRC table mismatch or an update of certain portions of the table, then there is a need to resynchronize the tables between master and slave. If the entire table is resent

then this will be the same result of expression 18. The CMC procedure in the DT3C will exchange 13 (equation 12 and 13) *ping/ ping response* packets to identify the corrupted BoI and one packet with the updated BoI. The total time the communication medium is occupied is (disregarding the inter frame space and the bit stuffing):

$$TableSyncTime_{DT3C} = \frac{13 \times 84 + 108}{1Mega} = 1.2 \text{ (ms)} \qquad (19)$$

Comparing with the option of resending the entire table (without any acknowledge), the DT3C uses 95.3% less the communication medium.

Considering the situation where there is corrupted data in more than one BoI. If the entire table is resent than all corrupted BoI will supposedly be eliminated (we will ignore the probability of having corrupted data again). In the DT3C, the CMC procedure will be repeated the same number of times that there are corrupted BoIs. This means that there is a maximum number of corrupted BoIs that makes the use of the DT3C worthwhile.

$$N_{MAX\_corruptedBoI} \frac{13 \times 84 + 108}{1Mega} = 277 \Rightarrow N_{MAX\_corruptedBoI} = \frac{277}{1.2} \approx 230 \qquad (20)$$

In this case (table size 10240) at a given instant, the DT3C will be more efficient than resending the entire table if there is less than 9% of corrupted BoIs.

The following graph shows the behaviour of the CMC algorithm as the table size increases and demonstrates the extent at which the CMC algorithm is more efficient than resending the entire table. As mentioned before, the number of simultaneous corrupted BoIs in a certain instant triggers the same number of CMC tree searches, hence the need to verify the maximum limit of corrupted BoIs.

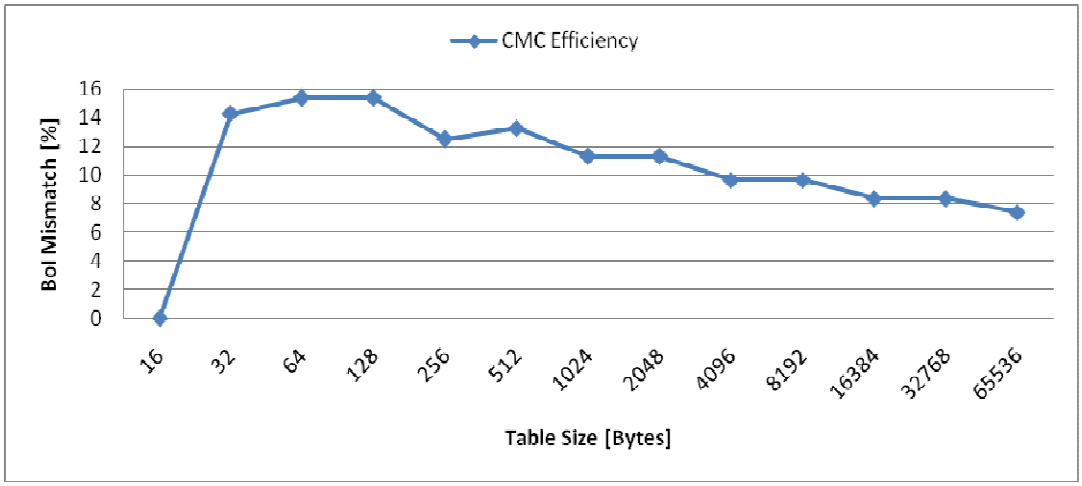

Figure 42 – Table size versus Corrupted BoIs

The efficiency of the CMC algorithm directly depends on the size allocated to the BoI in the config command (A2.1 - **TMU Management messages**). As previously seen, the size of the BoI is also affected by the size of the offset information. So, as the table size increases, the offset information gradually increases and as a result the BoI size decreases, thus contributing to a higher number of BoIs necessary to dispatch the table. It can be seen in figure 42 and in the table below that the CMC algorithm is not efficienct for small tables. The table below uses the values for each type of command established in the beggining of this subchapter and shows the same conclusion as figure 42.

Table Size (Bytes)	OI Size (Bytes)	BoI Size (Bytes)	Nº of Pkts or Nº of Boi	Table Dispatch (Second) With Ack	Table Dispatch (Second) DT3C	Table Synchronization Height	Table Synchronization Nº CMC	Table Synchronization DT3C (Seconds)	Nº of Boi Errors	%
16	1	5	4	0,000736	0,000432	2	4	0,000444	0	0,00
32	1	5	7	0,001288	0,000756	3	4	0,000444	1	14,29
64	1	5	13	0,002392	0,001404	4	6	0,000612	2	15,38
128	1	5	26	0,004784	0,002808	5	6	0,000612	4	15,38
256	2	4	64	0,011776	0,006912	6	8	0,00078	8	12,50
512	2	4	128	0,023552	0,013824	7	8	0,00078	17	13,28
1024	2	4	256	0,047104	0,027648	8	10	0,000948	29	11,33
2048	2	4	512	0,094208	0,055296	9	10	0,000948	58	11,33
4096	2	4	1024	0,188416	0,110592	10	12	0,001116	99	9,67
8192	2	4	2048	0,376832	0,221184	11	12	0,001116	198	9,67
16384	2	4	4096	0,753664	0,442368	12	14	0,001284	344	8,40
32768	2	4	8192	1,507328	0,884736	13	14	0,001284	689	8,41
65536	3	3	21846	4,019664	2,359368	15	16	0,001452	1624	7,43

Table 9 – CMC Algorithm Efficiency

The use of parallel CMC algorithms could improve the efficiency of the DT3C. However, this solution alone will not improve the efficiency for small tables. Other mechanisms must be implemented so that the use of parallel CMC algorithms is worthwhile.

7 Conclusion and Future Work

The research community in this area has presented many solutions to improve the dependability in CAN networks. The benefits of redundancy, topology change, fault detection and isolation are used to a certain extent in each of these solutions. Some only make use of redundancy or fault detection and isolation, while others include all of the methods in order to maximize the networks dependability. The Dynamic Topology Management in CAN is a complex concept that makes use of the three methods stated above. The DTM is still in its initial stage of research. Its complexity mostly resides in the Topology Management Unit, which is the main focus of this dissertation. The TMU has to manage the redundancy of the communication medium, its own redundancy and also the access of each Network Switch Unit to the available communication medium.

Research was done with the purpose of occupying the communication medium as little as possible with the TMU management message exchange, consequently reducing traffic interference between nodes located in each NSU. In safety critical networks, the availability of the communication medium is fundamental and therefore the TMU management information should occupy the medium as little as possible. Special attention was given to the possibility of taking advantage of the CAN transceivers filtering capability and transmitting management information at higher frequency. Additional and dedicated hardware had to interface each communication medium in order to transmit and receive data at much higher data rates. This idea was left for future research of the DTM network. Instead, the exchange of TMU management information makes use of standard CAN messages.

Having established how the TMU will communicate with the NSUs and TMU slaves, a protocol was defined to provide an efficient topology management. The internal functionality of the TMU was investigated and defined. The commands which are embedded in the data frame payload were defined for all possible TMU and NSU management issues. Some of the commands have fields that are currently not used but could be used in future releases without compromising older releases. Rules were established for the TMU slave to take control over the DTM network. A conflict resolution was defined in situations where more than one master was simultaneously present. The first master to send a *ping* command takes control of the DTM. The identifier of each TMU master acts as a final conflict resolution in case both masters try to send simultaneously a *ping* command. Another issue that had to be solved was the fact that the TMU can have several CAN controllers and needs a method to process the incoming messages. The trick is that the TMU slaves and the NSUs only send information upon a request of the TMU master. Therefore, the TMU master always knows which type of command it is expecting after sending a request to a TMU slave or NSU. Upon the reception of the expected command on all communication mediums (a timeout is established if the command was not received on all communication mediums), a majority vote scheme is used to retrieve the commands content, since it is not possible to use the content of all incoming messages. All packets received by the TMU master that do not correspond to the expected command are simply ignored. The TMU always transmits information on all communication mediums. Hence, transmitted messages do not pose a problem, since the data is sent to all CAN controllers at the same time. Finally, the potentiality of a DTM system – TMUCAM software was briefly described. This software should be able to analyze, control and monitor the status of each entity.

After describing the internal functionalities in the TMU there was a necessity to identify the main hardware blocks which are responsible for TMU behavior. Although these blocks were developed, the main focus in this dissertation is the DT3C module. The rest of the TMU blocks are secondary and therefore not described in detail in this dissertation.

The TMUs uses standard CAN messages to establish synchronization and therefore it was necessary to define a procedure that allows bandwidth economy. The Data Table Content Consistency Checker - DT3C has the objective of guaranteeing consistency between the TMU master and its slaves with low bandwidth occupation compared to traditional

communication protocols. Results show that the DT3C occupies half the bandwidth when compared to a communication that uses an Acknowledge for every received packet. In terms of resynchronization the traditional method normally implies that the master resends the entire data table. The DT3C uses a search tree algorithm and CRC calculations to compare and find the portion of corrupted data in the slaves. One of the conclusions was that the number of packets exchanged between master and slave (to find the corrupted data) depends on the height (level) of the tree, which has a logarithmic growth as the number of BoIs increase. Hence, the packet DT3C is really bandwidth efficient when compared to the option of resending the entire data table. The results obtained for the situation where there were more than one corrupted BoI at the same time was not the most satisfactory, since it is only efficient if there is less than 9% of corrupted BoIs as shown in the example. An implementation in FPGA confirmed the possibility of a purely hardware design, whereas some matlab simulations confirmed the search tree algorithm's ability to encounter corrupted data table.

7.1 Future Research

The dissertation focused in the general issues of the TMU and in a specific problem: the data replication consistency scheme. Various suggestions are made for future improvements and development of the TMU.

TMU

The main objective of this work was to identify and propose a stable TMU with basic functionalities that would allow a simple yet partially efficient management of the DTM. However, there is still plenty of space for improvements in the TMU.

The CAN transceivers demonstrate a good frequency filtering capability, so there could be a chance in investigating the possibility of using frequency multiplex on the same communication medium. The TMU could communicate with each other and with the NSUs in a higher frequency without disturbing the node communication.

The scheduler implemented in the TMU is a hardware FSM. Adding new functionalities to the scheduler could be a little complicated since this often means that there must be a total redesign of the entire FSM. This job could turn out to be easier if it would be done in software. So one of the suggestions for improving the TMU could be implementing a SOC (uC, RAM, UART, CRC generators and CAN controllers), and have a software scheduler run in the uC. The following suggestions for scheduler improvements are:

- *Ping* NSUs for partial media usage;
- Be able to change CAN baud rate during DTM operation;
- Internal redundancy in the TMU to guarantee fail safe: this was not considered during this dissertation;

DT3C

The CMC algorithm used may not be the most efficient among the various tree search algorithms. There is plenty of research in the software area related to data structure search algorithms. Future work could be done in the research of the most appropriate and efficient search algorithm for the CMC.

Another improvement to the CMC could be in the use of more information transport in the packet payload during the CMC tree search algorithm. This could imply that each packet exchanged between master and slave could carry more than one CRC calculation, thus achieving the tree leaf in less iterations (and with less package exchange). The idea of using parallel CMC tree search algorithm and data field space reallocation seems to be a promising solution to enhance the efficiency of the DT3C.

During bring up (first time there is a table dispatched to the slave) a packet omission is easily detected because all CRC memory is set to zero. The lack of a packet reception for a certain BoI means that there will be no CRC BoI in the CRC memory and no BoI in the

table memory at the location specified by the OI. This will lead to a CRC table mismatch, which will trigger the CMC procedure. This procedure will verify that there is a mismatch between the CRC calculated from the BoI read from the table memory and the CRC value (will be zero) stored in the CRC memory. The ICV procedure could identify packet omissions during the first table dispatch and immediately request for an update, instead of running the CMC procedure. A problem arises during normal operation when the master updates one or more BoIs. In this case, if there is a packet omission this will not be detected by the ICV procedure but by the CMC procedure. The reason is simple, slave already has data and its CRC calculation stored for each offset information in the CRC and table memory, correspondingly. The ICV will not encounter any mismatch, leaving the CMC procedure to find the corrupted data (which is a BoI that was not updated because of a packet omission). This means that the ICV will only identify packet omission during bring up. During normal operation its only functionality will guarantee that the values stored in the CRC memory (CRC BoIs) correspond to the CRC calculation of each BoI stored in the table memory, but will not identify packet omissions during table update. The packet omission detection could be enhanced. A suggestion is that the master can send information indicating which BoIs it has intention of updating before dispatching the updated BoIs. Probably the slave should send an acknowledge for this information since it is critical that the master knows that the information was successfully received. Before the master sends the BoIs, the slave can perhaps zero the CRC memory for these updated BoIs. In this case, if there is a CRC table mismatch due to packet omission, then the ICV will detect and request the missing BoI(s).

A last suggestion is to improve the DT3C implementation in terms of number of clock cycles necessary in operation, keeping in mind the number of logic used in the implementation.

Appendix

A1. CAN Communication Media Experiment

The first possibility is to analyze the feasibility of sending a predefined sequence of high frequency impulses (similar idea to [18]) through a CAN transceiver during a specific bit in a CAN frame without interfering with the communication between the various nodes which are connected to the bus.

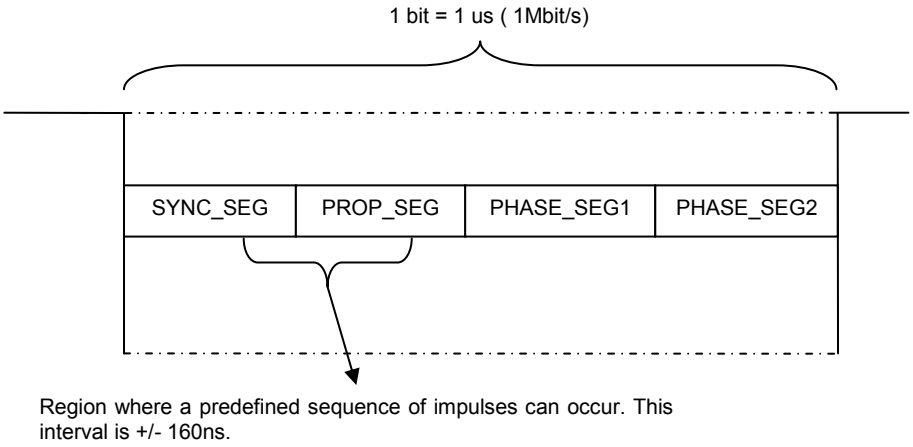

Figure 43 – Bit Time

High frequency is defined as a signal that has at least twice the frequency of the maximum data rate 1Mbit/s. The following setup was used for testing the transceivers frequency response:

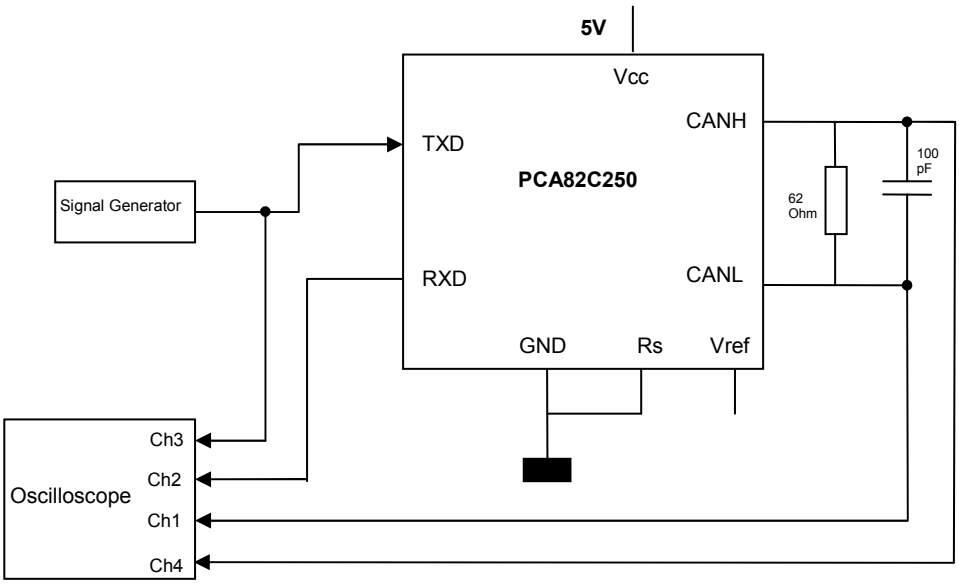

Figure 44 – Test Setup

The transceiver used in this test was a PCA82C250, which has (datasheet 13/01/2000) the following specifications:

- Maximum differential input resistance = 100 KOhm
- Maximum differential input capacitance = 10 pF

This means that each node that is attached to the bus introduces a 10pF and a 100KOhm in parallel with the total bus impedance, as shown in Figure 45 – **Total Impedance**:

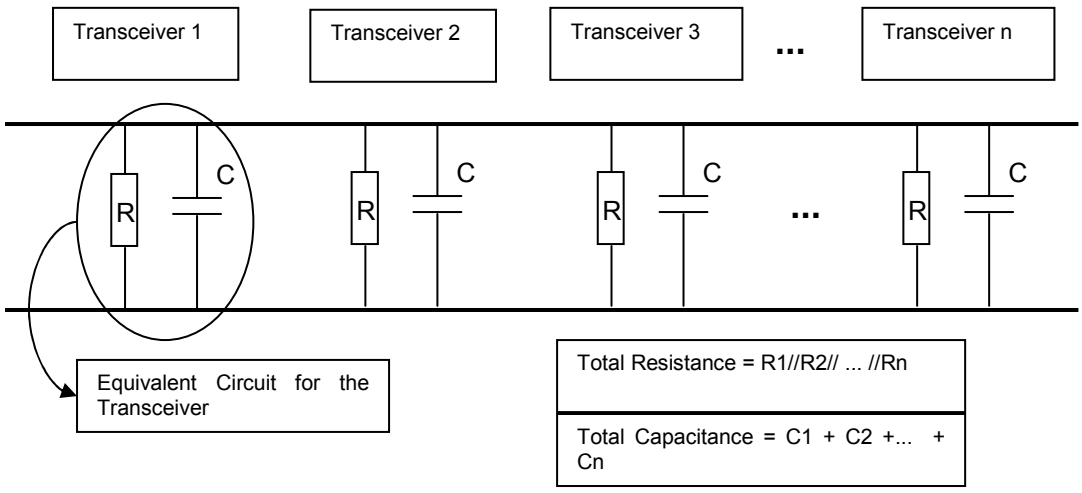

Figure 45 – Total Impedance

There can be 1nF / 10pF = 100 nodes attached to the bus for the case where the load capacitance was assumed to be 1nF. For 100 nodes, the total equivalent resistance is 58.72 Ohms. The following table gives us the equivalent values depending on the number of nodes.

Nodes	Total Resistance	Total Capacitance
100	58.72 Ohms	1 nF
33	60.76 Ohms	330 pF
10	61.62 Ohms	100 pF

Table 10 – Nº of Nodes versus Resistance/Capacitance

Various tests were made taking only into account the capacitance value. The following table and figures resumes the tests (for the various loads):

Nº of Nodes	Load	Maximum Frequency
10	R= 62 Ohm, C= 100 pF	6.25 MHz
33	R= 62 Ohm, C= 330 pF	Not verified
100	R= 62 Ohm, C= 1 nF	3.125 MHz

Table 11 – Nº of nodes versus Frequency

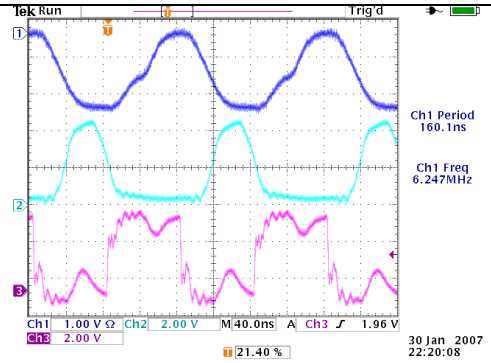

Figure 46 – TXD @ 6.25MHz

Ch1 -> CAN bus (Differential probe)

Ch2 -> RXD

Ch3 -> TXD

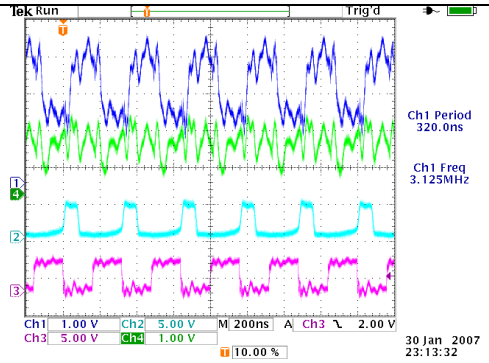

Figure 47 – TXD @ 3.125MHz

Ch1 -> CANH

Ch4 -> CANL

Ch2 -> RXD

Ch3 -> TXD

The results shown above clearly demonstrate that for a CAN network working at 1 Mbit/s (where each bit is 1us) the use of a CAN transceiver is not appropriate for the transmission of high frequency impulses. In this situation the impulse can be at the best 80 ns in width (for only 10 nodes attached to the bus) or 160 ns (for 100 nodes).

After some tests it can be concluded that there are three possible solutions for the TMU:

Solution 1. Using the transceiver to generate a 160 ns impulse (worst case). This impulse should be transmitted during a recessive bit in a frame. Probably the best bit for this is the CRC delimiter or the ACK delimiter, or one of the end of frame bits. The region where the impulse could be transmitted without disobeying the synchronization rules of the CAN standard (and without disrupting the frame structure) is pointed out in figure 4 and explained below [2]:

HARD SYNCHRONIZATION and RESYNCHRONIZATION are the two forms of SYNCHRONIZATION. They obey the following rules:

I. Only one SYNCHRONIZATION within one bit time is allowed.

II. An edge will be used for SYNCHRONIZATION only if the value detected at the previous SAMPLE POINT (previous read bus value) differs from the bus value immediately after the edge.

III. HARD SYNCHRONIZATION is performed whenever there is a 'recessive' to 'dominant' edge during BUS IDLE.

IV. All other 'recessive' to 'dominant' edges fulfilling the rules 1 and 2 will be used for RESYNCHRONIZATION with the exception that a node transmitting a dominant bit will not perform a RESYNCHRONIZATION as a result of a 'recessive' to 'dominant' edge with a positive PHASE ERROR, if only 'recessive' to 'dominant' edges are used for resynchronization.

Solution 2. Another solution would be to develop a special hardware which can produce and detect smaller impulses (around 50 ns). This would allow the TMU's to communicate without interfering with the nodes, as the transceiver filters out these signals. The drawback (besides developing the hardware) is the maximum frequency of the FPGA which would limit the sampling rate and therefore the impulse width. The following scheme is suggested:

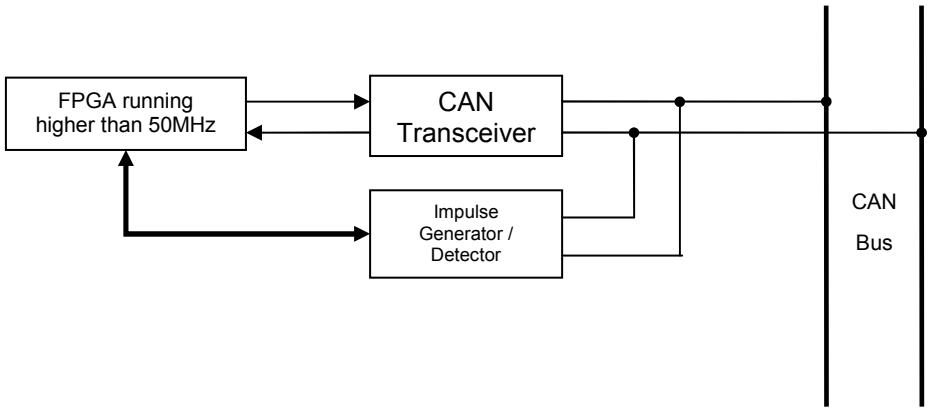

Figure 48 – CAN Transceiver Test Setup

Solution 3. Finally, the last solution should be the use of standard CAN messages for the TMU fault detection and communication.

A2. TMU ARQUITECTURE

1. TMU Management messages

In order to communicate with each TMU, a set of commands were defined in the data field of a data frame. Although the identifier is mentioned in the description of each command, it will not be represented in the command mappings described for each command. The following commands were defined for the communication between TMU master and each TMU slave:

- CONFIG(ACK, data_type, Lf, data)

Data Field Byte	0		1			2	3	4	5	6	7
Bit	7..1	0	7.. 5	4 .. 1	0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	ACK	data_type	Not used	Lf	data	data	data	data	data	data

Where,

  - CMD – Command 0x12;
  - TMadd – target TMU Master address is mapped in the ID field of the CAN frame (not represented in the command byte map);
  - ACK – if asserted, then target must reply with an acknowledge (optional);
  - Data_type – identifies which data is being sent;
    - 000 -> NSU table data
    - 001 -> reconfiguration table data
    - 010 -> Topology table data
    - 011 -> ConfigRegBank data
    - 100 -> Swap Current NSU table (optional)
  - Lf – last frame of data (optional);
  - Data – the data from one of the sources defined by data_type.

- PING(MBview)

Data Field Byte	0		1	2	3	4	5	6	7
Bit	7..1	0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	Not used	MBview	crcSt	crcH	crcL	Not used	Not used	Not used

When a slave receives this command, it must reply with a PING_RESPONSE. Where,

  - CMD – Command 0x14;
  - TSadd – target TMU Slave address is mapped in the ID field of the CAN frame (not represented in the command byte map);
  - MBview – Bus status from the master's point of view;
  - crcSt – this has CMC information;
  - crcH – most significant byte of the 16 bit CRC;
  - crcL – least significant byte of the 16 bit CRC.

- PING_RESPONSE(SBview)

Data Field Byte	0		1	2	3	4	5	6	7
Bit	7..1	0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	Not used	SBview	crcSt	crcH	crcL	Not used	Not used	Not used

Where,

- CMD – Command 0x16;
- TMadd – target TMU Master address is mapped in the ID field of the CAN frame (not represented in the command byte map);
- SBview – Bus status from the slave's point of view;
- crcSt – this has CMC information;
- crcH – most significant byte of the 16 bit CRC;
- crcL – least significant byte of the 16 bit CRC.

- ACKNOWLEDGE(Tadd, CCore)

Data Field Byte	0		1		2	3	4	5	6	7
Bit	7..1	0	7.. 3	2 .. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD			NTadd		CCore	Not used	Not used	Not used	Not used

Where,

- CMD – Command 0x04;
- TMadd – target TMU Master address is mapped in the ID field of the CAN frame (not represented in the command byte map);
- NTadd – identifies the origin of the acknowledge (NSU or TMU address). This address is mapped in byte[1] (bit 2..0) and byte[2] of the data field.
- CCore – identifies the command that originated this acknowledge and is mapped in byte[3] of the data field.

2. TMU and NSU Management messages

In order to communicate with each NSU, a set of commands were defined in the data field of a data frame. Although the identifier is mentioned in the description of each command, it will not be represented in the command mappings described for each command. The following commands were defined for the communication between TMU and NSU:

- SET_ADDRESS(TNadd, NNad, ACK, bus1, …, busM)

Data Field Byte	0		1		2	3	4	5	6	7
Bit	7..1	0	7.. 3	2..0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	ACK	Not used	NNadd		Bus#	Not used	Not used	Not used	Not used

Where,

- CMD – Command 0x02;
- TNadd – default address. Mapped in the ID field of the CAN frame (not represented in the command byte map);
- NNadd – new address. Mapped in byte[1] (bit 2..0) and byte[2];
- ACK – if asserted, then target must reply with an acknowledge. Mapped in bit 0 of byte[0] of the data field;
- Bus# – identifies which bus(ses) the ACK will be sent (this functionality is not implemented, ACK is sent to all buses). Mapped in byte[3].

- ACKNOWLEDGE(Tadd, CCore)

  This command is explained in 4.4 (also used by the TMU slave).

- OPEN_SWITCHES(Nadd, tbus1, …, tbusM, ACK, rbus1, …, rbusM)

Data Field Byte	0		1	2	3	4	5	6	7
Bit	7..1	0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	ACK	tbus#	rbus#	Not used	Not used	Not used	Not used	Not used

Where,

- CMD – Command 0x06;
- Nadd – target NSU address, mapped in the ID field of the CAN frame (not represented in the command byte map);
- tbus# – bit flags which identify the disconnected buses. This is mapped in byte[1] of the data field;
- ACK – if asserted, then target must reply with an acknowledge. Mapped in bit 0 of byte[0] of the data field;
- rbus# – identifies which bus(ses) the ACK will be sent. This is mapped in byte[2] of the data field;

- CLOSE_SWITCHES(Nadd, tbus1, …, tbusM, ACK, rbus1, …, rbusM)

Data Field Byte	0		1	2	3	4	5	6	7
Bit	7..1	0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	ACK	tbus#	rbus#	Not used	Not used	Not used	Not used	Not used

Where,

- CMD – Command 0x08;
- Nadd – target NSU address, mapped in the ID field of the CAN frame (not represented in the command byte map);
- tbus# – bit flags which identify the connected buses. This is mapped in byte[1] of the data field;
- ACK – if asserted, then target must reply with an acknowledge. Mapped in bit 0 of byte[0] of the data field;
- rbus# – identifies which bus(ses) the ACK will be sent. This is mapped in byte[2] of the data field.

- Matrix_WRITE(Nadd, ACK, bus1, …, busM)

Data Field Byte	0		1	2	3	4	5	6	7
Bit	7..1	0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	ACK	bus#	Not used	Not used	Not used	Not used	Not used	Not used

Where,

- CMD – Command 0x0A;
- Nadd – target NSU address, mapped in the ID field of the CAN frame (not represented in the command byte map);
- ACK – if asserted, then target must reply with an acknowledge. Mapped in bit 0 of byte[0] of the data field;
- bus# – identifies which bus(ses) the ACK will be sent. This is mapped in byte[1] of the data field.

- Matrix_READ(Nadd, rbus1, …, rbusM)

Data Field Byte	0		1	2	3	4	5	6	7
Bit	7..1	0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	Not used	rbus#	Not used	Not used	Not used	Not used	Not used	Not used

Where,

- CMD – Command 0x0C;
- Nadd – target NSU address, mapped in the ID field of the CAN frame (not represented in the command byte map);
- rbus# – identifies which bus(ses) the ACK will be sent. This is mapped in byte[1] of the data field.

- Matrix_DATA(Tadd, Nadd, Switches_Data)

Data Field Byte	0		1		2	3	4	5	6	7
Bit	7..1	0	7.. 3	2 .. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0	7.. 0
	CMD	Not used		Nadd		Size	Data			

Where,

- CMD – Command 0x0E;
- Tadd – target TMU address, mapped in the ID field of the CAN frame (not represented in the command byte map);
- Nadd – NSU address, mapped in byte[1] (bit 2..0) and byte[2];
- Size – size of matrix. Mapped in byte[3];
- Data – the command that originated this acknowledge. Mapped in byte[4] to byte[7].

3. Serial Communication Protocol

The TMU serial port has a fixed configuration:

Fixed Parameters	Value
Baud Rate	115200
Data	8 bits
Parity	None
Stop	1 bit
Flow Control	None

The most significant data nibble (MSN) is used to define a command, while the least significant data nibble (LSN) contains information. All data sent over the serial communication to the TMU must have the following structure:

Bit	7	6	5	4	3	2	1	0
	Command Code				Data			

The following table describes the command codes sent to the TMU:

Command Code	Bit 7		Bit 6	Bit 5	Bit 4
		Last Byte			
Reset	0	1	0	0	0
NSU Table	0	1	0	0	1
Reconfig. Table	0	1	0	1	0
Topology Table	0	1	0	1	1
ConfigRegBank	0	1	1	0	0
QoSStartStream	0		1	0	1
QoSStopStream	0		1	1	0
ReadQoSRegBank	0		1	1	1
TbReader	1		1	1	1

When the TMU receives a ReadQoSRegBank command, it will send a fixed number of bytes containing information relative to Qos information. The TMU will respond to a TbReader, sending all information in the tables and configRegBank. As for the NSU, Reconfig, Topology and ConfigRegBank tables commands, they are sent all without acknowledge and verified at the end with a CRC code.

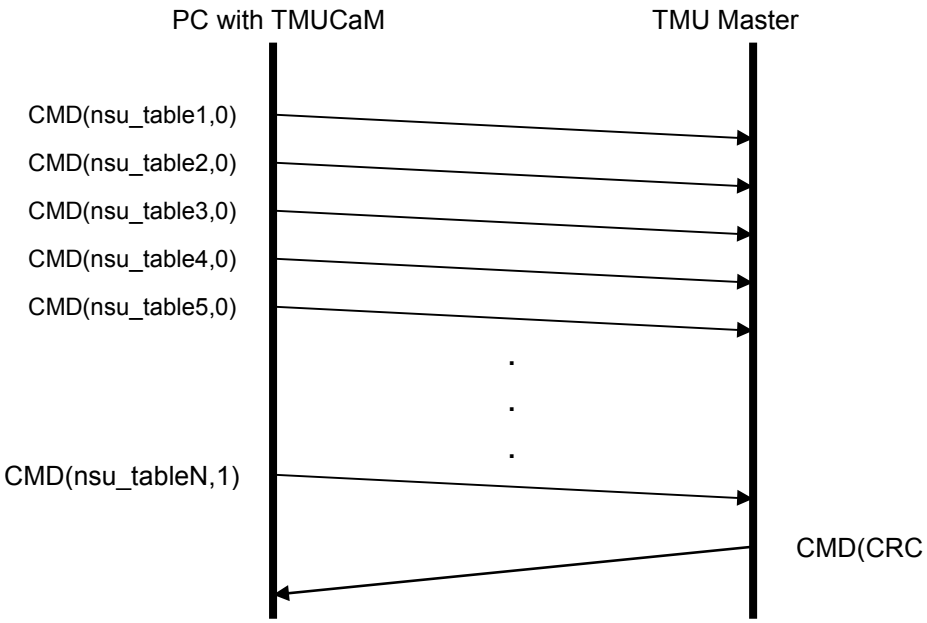

Figure 49 – Communication between TMU and PC

NSU, Reconfiguration and Topology Table

The following tables are organized:

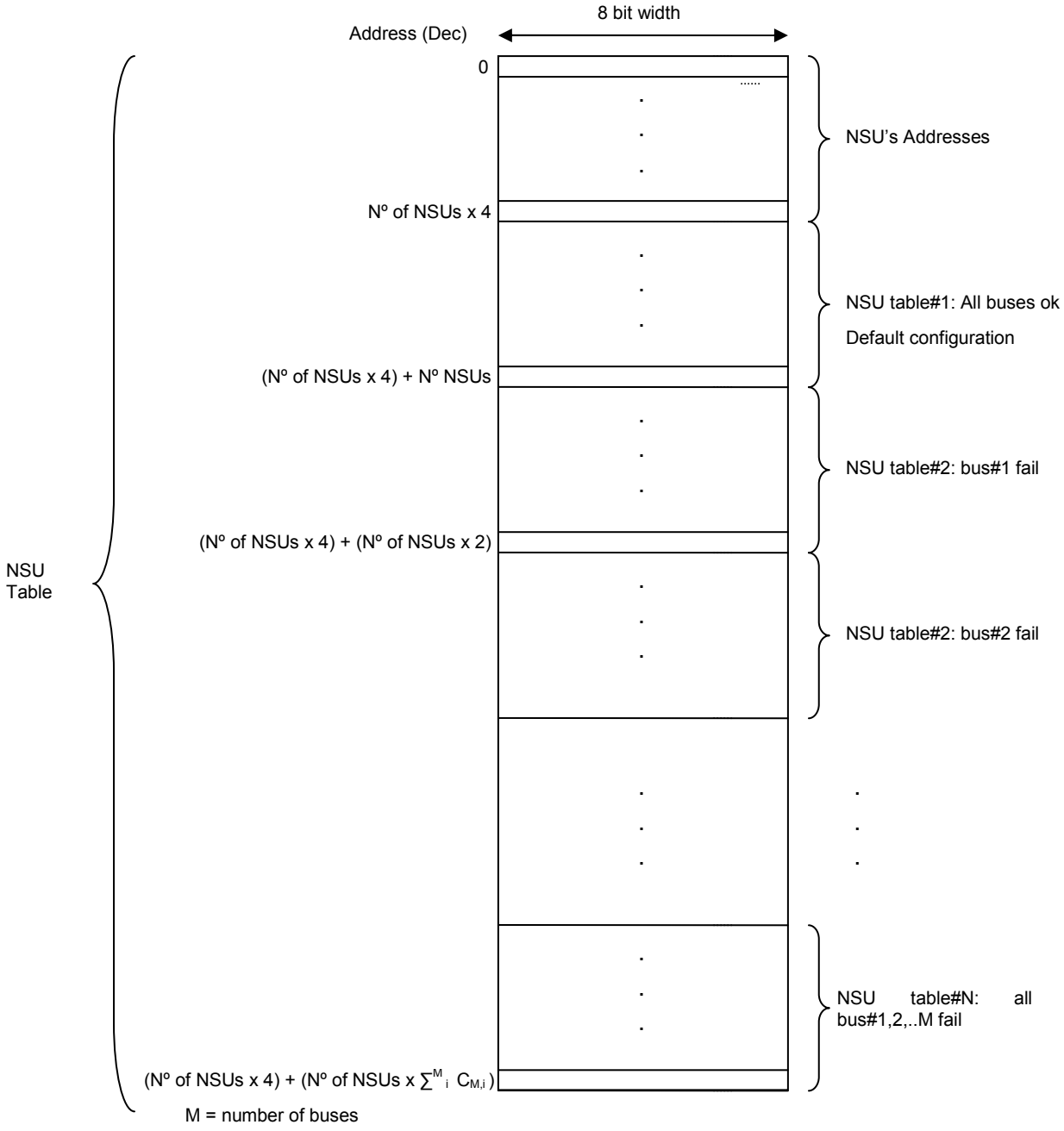

Figure 50 – NSU Table

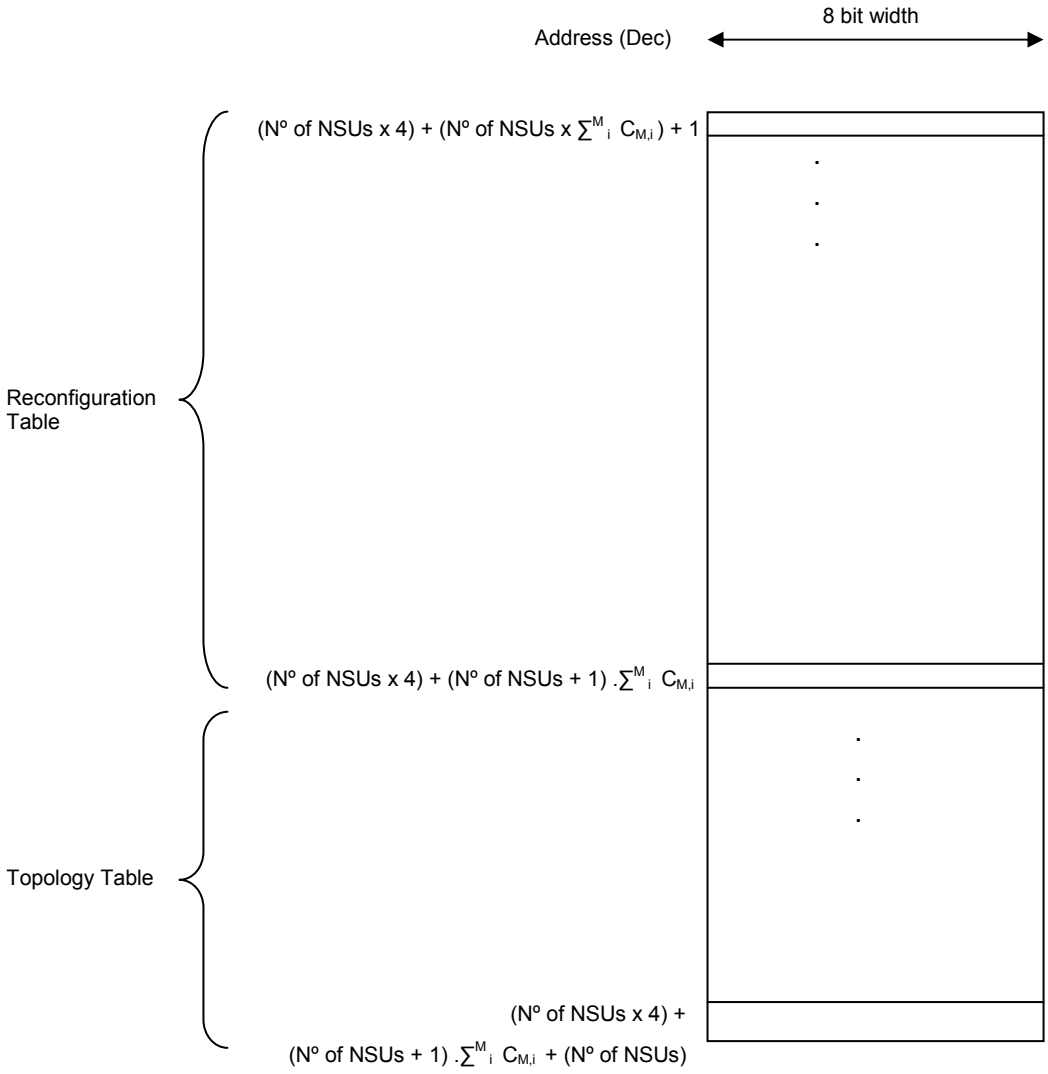

Figure 51 – Reconfiguration and Topology Table

4. ConfigRegBank

All registers are read and write.

The following table shows which are the default values of a TMU after hardware reset:

Configurable Parameters	Default Value
Mode	Slave
CAN ID	0b00000000010
CAN Baud Rate	1MBit/s

Table 12 – TMU default values

TMU Configuration Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NCB 2	NCB 1	NCB 0	0	0	0	MODE	CONF

Table 13 – TMU Configuration Register implementation

CONF: if set to '1', indicates that the TMU has been configured by the TMUCAM. Upon TMU initialization or hardware reset, this bit is set to '0'.

MODE: if set to '1', indicates that the TMU is Master. Upon TMU initialization or hardware reset, this bit is set to '0' and therefore TMU is in slave mode.

NCB[2..0]: defines the number of CAN buses. The following sequence of bits are used:

NCB 2	NCB 1	NCB 0	
0	0	0	TMU is not configured. Reset
0	0	1	One CAN bus available in system.
0	1	0	Two CAN buses available in system.
0	1	1	Three CAN buses available in system.
1	0	0	Four CAN buses available in system.
1	0	1	TMU currently only supports a maximum of 4 buses. Other values defined in the NCB bits are parsed as 4 buses.
1	1	0	
1	1	1	

NNSU Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
NSU 7	NSU 6	NSU 5	NSU 4	NSU 3	NSU 2	NSU 1	NSU 0

Table 14 – NNSU Register implementation

NSU [7..0]: Defines the number of NSUs in the system. Upon TMU initialization or hardware reset, these bits are set to '0'.

CANCONF1 Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
SJW 1	SJW 0	BRPS 5	BRPS 4	BRPS 3	BRPS 2	BRPS 1	BRPS 0

Table 15 – CANCONF1 Register implementation

BRPS [5..0]: The baud rate prescaler defines the clock frequency division. Upon TMU initialization or hardware reset, these bits are set to ‘0’.

SJW [1..0]: The synchronization jump width bits define the value in time quanta. Upon TMU initialization or hardware reset, these bits are set to ‘0’.

CANCONF2 Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	0	IDEXT

Table 16 – CANCONF2 Register implementation

IDEXT: If set to ‘1’ then CAN extended identifier is enabled. Upon TMU initialization or hardware reset, this bit is set to ‘0’.

TSEG Register

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
TS2 2	TS2 1	TS2 0	TS1 4	TS1 3	TS1 2	TS1 1	TS1 0

Table 17 – TSEG Register implementation

TS1 [4..0]: Defines the length -1 (in time quanta) of the CAN bit time segment 1. Upon TMU initialization or hardware reset, these bits are set to ‘0’.

TS2 [2..0]: Defines the length -1 (in time quanta) of the CAN bit time segment 2. Upon TMU initialization or hardware reset, these bits are set to ‘0’.

A3. DT3C Architecture

1. External Interface – DT3C IF (Level 0)

This section describes the 4 interfaces of the top level block: System I/F, Debug I/F, Status I/F, Config I/F, Comm I/F. All signals

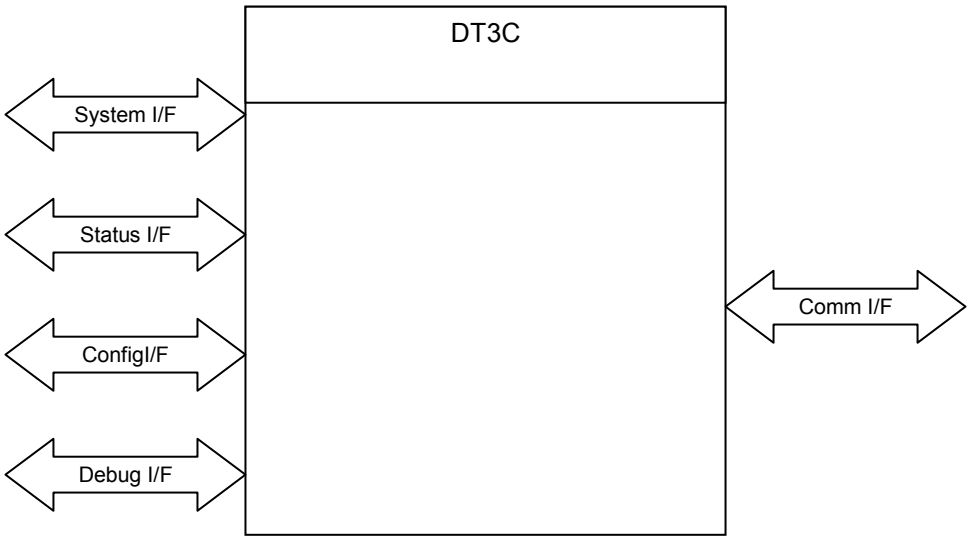

Figure 52 – DT3C external interfaces

1.1. System I/F

The following table describes the global signals used in the entire design.

Name	Edge	Dir	Frequency	Description
t_clk_i	r	I	5.0000 MHz	Master Clock
s_rst_i	r	I	-	Master Reset
s_en_i	r	I	-	Master enable

Table 18 – Global Signals of System I/F

1.2. Debug I/F

The following table describes the debug signals of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
s_test_io	16	io	t_clk_i	-	16'hZ	Test Port. A general purpose 16-bit bi-directional port is provided for debug/test issues.

Table 19 – Debug I/F

1.3. Status I/F

The following table describes the status signals of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
s_alarm_o	1	O	t_clk_i	H	1'h0	Indicates that a system failure occurred and all DT3C operations are not stopped.
s_opOk_o	1	O	t_clk_i	H	1'h0	Indicates normal operation.
s_masterMode_o	1	O	t_clk_i	H	1'h0	Indicates that the DT3C is running in master mode.
s_slaveMode_o	1	O	t_clk_i	H	1'h0	Indicates that the DT3C is running in slave mode.

Table 20 – status I/F

1.4. Config I/F

The following table describes the configurational signals of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
s_rstMain_i	1	i	t_clk_i	H	-	Reset main Finite state machine.
s_master_i	1	i	t_clk_i	H	-	Define module as master.
s_slave_i	1	i	t_clk_i	H	-	Define module as slave.

Table 21 – Config I/F

1.5. Comm I/F

The following table describes the communication signals of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
s_dataRxMed_i	DF	I	t_clk_i	-	-	Communication data reception channel.
s_dataRxMem_i	8	I	t_clk_i	-	-	External memory data reception
s_addrInit_i	A	I	t_clk_i	-	-	Initial external offset memory address
s_tblEnd_i	1	I	t_clk_i	H	-	End of table transmission for master's initial table upload.
s_rxCrcTblAck_i	1	I	t_clk_i	H	-	Indicates that data received is a CRC table acknowledge.
s_rxCrcTblNack_i	1	I	t_clk_i	H	-	Indicates that data received is a CRC table which was not acknowledged.
s_rxSrchCrcChck_i	1	I	t_clk_i	H	-	Indicates that data received is a search CRC check between master and slave.
s_rxRdy_i	1	I	t_clk_i	H	-	This signal is asserted when all output data is stable for reception.
s_rxVldSrchCrc_i	1	I	t_clk_i	H	-	Receive valid CRC comparison
s_dataTxMed_o	DF	O	t_clk_i	-	16'hZZ	Medium transmit data bus
s_addrMem_o	A	O	t_clk_i	-	Z	Table Address bus
s_dataTxMem_o	8	O	t_clk_i	-	Z	Table transmit data bus
s_rdMem_o	1	O	t_clk_i	-	0	Read table memory signal
s_wrMem_o	1	O	t_clk_i	-	0	Write table memory signal
s_noSlvRsp_o	1	O	t_clk_i	-	0	No slave response alarm signal
s_txRdy_o	1	O	t_clk_i	-	0	This signal is asserted when all input data is stable for transmission.

s_rxEnd_o	1	O	t_clk_i	-	0	This signal is asserted to indicate end of data reception.
s_txCrcTblNack_o	1	O	t_clk_i	-	0	Indicates that data transmitted is a CRC table which was not acknowledged.
s_txCrcTblAck_o	1	O	t_clk_i	-	0	Indicates that data transmitted is a CRC table which was acknowledged.
s_srchCrcChck_o	1	O	t_clk_i	-	0	Indicates that data being sent is a Search CRC Check.
s_txVldSrchCrc_o	1	O	t_clk_i	-	0	Transmit a valid CRC comparison

Table 22 – Comm I/F

2. Internal Functions

The DT3C is subdivided internally by 6 modules.

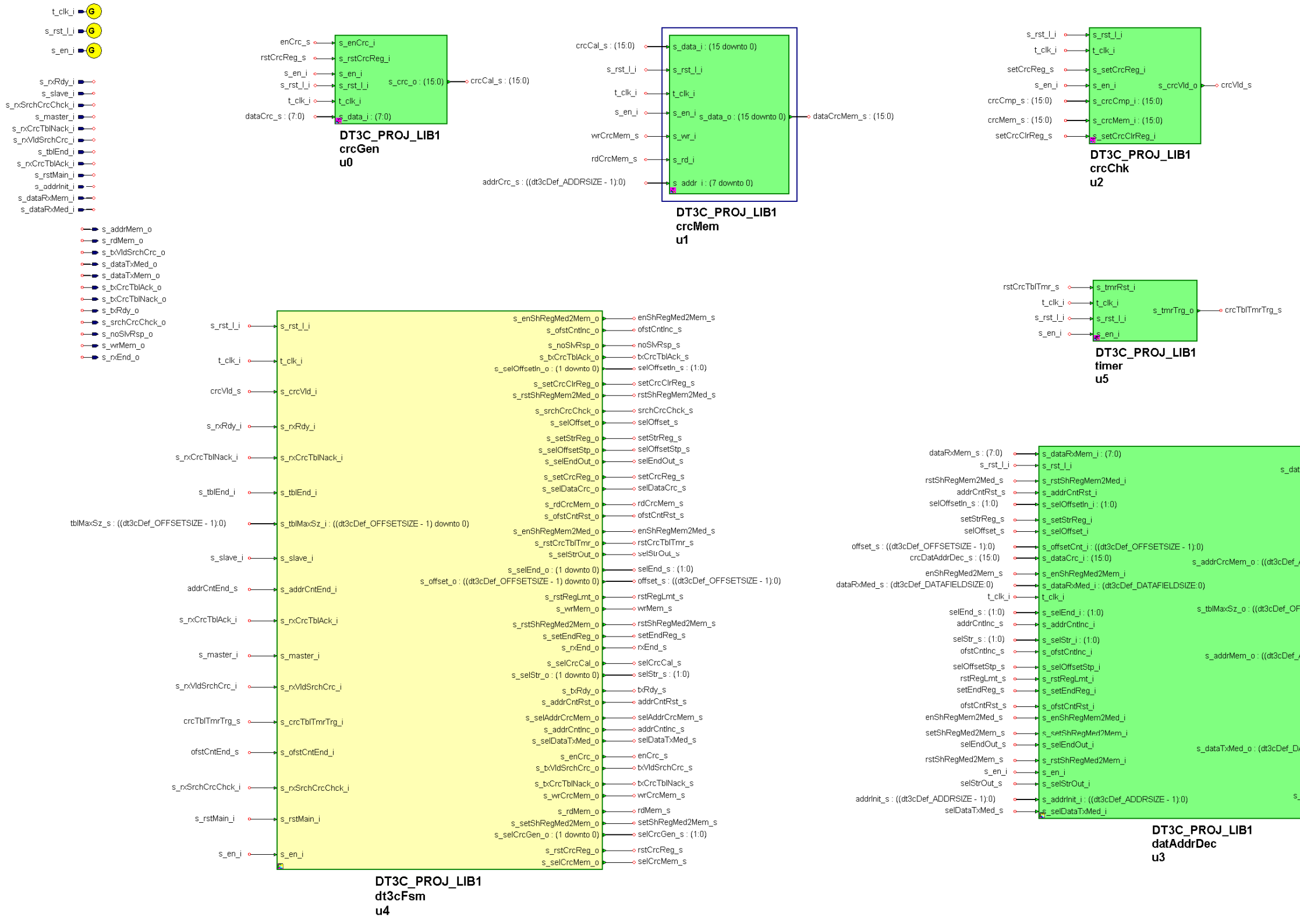

Figure 53 – DT3C internal modules

2.1. Cyclic Redundancy Check Generator – crcGen IF (Level 1)

2.1.1. Overview

The crcGEN module is a bytewise or parallel CRC generator. There are basically two types [22] of hardware solutions for CRC computation. A traditional hardware solution is the serial or bitwise CRC which consists in a Linear Shift Register (LSR) with serial data feed. The bytewise or parallel CRC hardware solution has some advantage because it has a speed-up factor between 4 and 6 when using a parallelism of 8. The parallel hardware solution basic blocks can be seen in Figure 54 – **CRC Generator**. Another method is using a CRC lookup table to calculate the CRC. This method was not adopted because the look up table has to be inputted and the previous solution is simpler to implement.

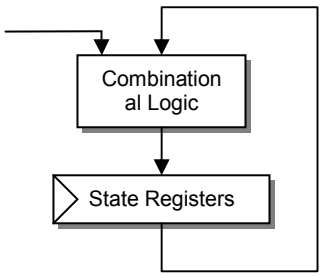

Figure 54 – CRC Generator

2.1.2. Interface Description

The following table describes the clock(s) of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
t_clk_i	1	I	t_clk_i	-	-	Master Clock
s_rst_l_i	1	I	t_clk_i	L	-	Master Reset
s_en_i	1	I	t_clk_i	H	-	Master enable
s_enCrc_i	1	I	t_clk_i	H	-	Enable CRC logic
s_rstCrcReg_i	1	I	t_clk_i	H	-	Reset CRC internal registers
s_data_i	8	I	t_clk_i	-	-	Data input – data used to calculate CRC
s_crc_o	16	o	t_clk_i	-	16'hFFFF	CRC output

Table 23 – crcGen Interface

2.1.3. Implementation

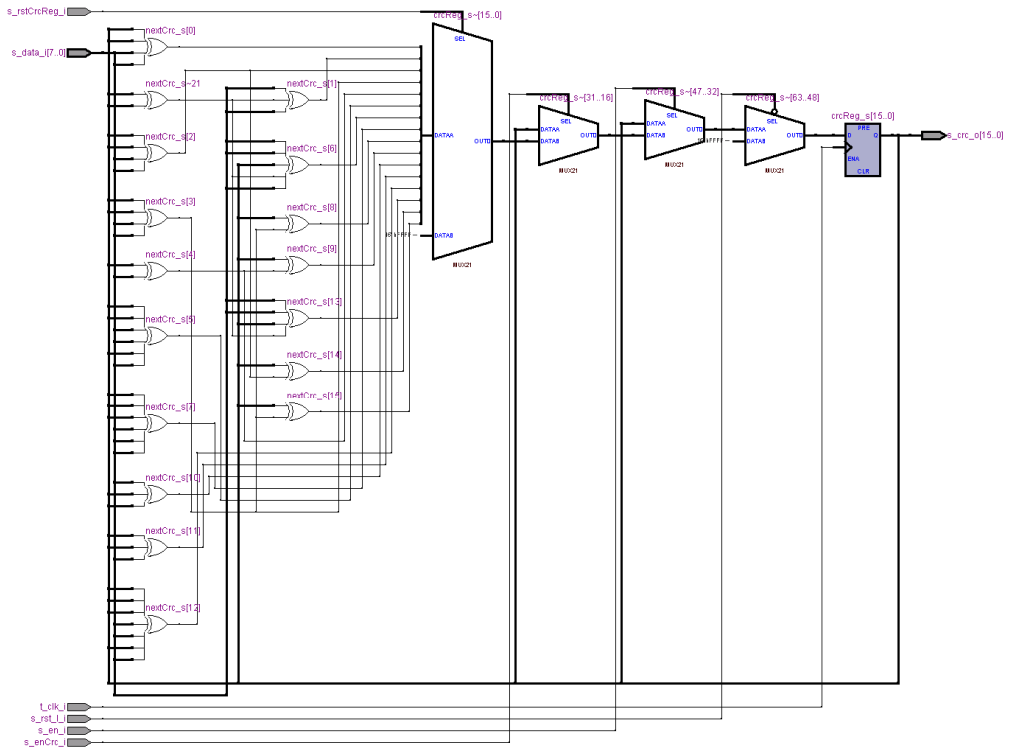

Figure 55 – crcGen RTL

2.2. Cyclic Redundancy Check Memory – crcMem IF (Level 1)

2.2.1. Overview

In order to implement a generic DT3C which is independent of any supplier or architecture, the internal memory which contains the CRC calculations must be implemented as a general memory and must not make use of specific FPGA memory functionalities.

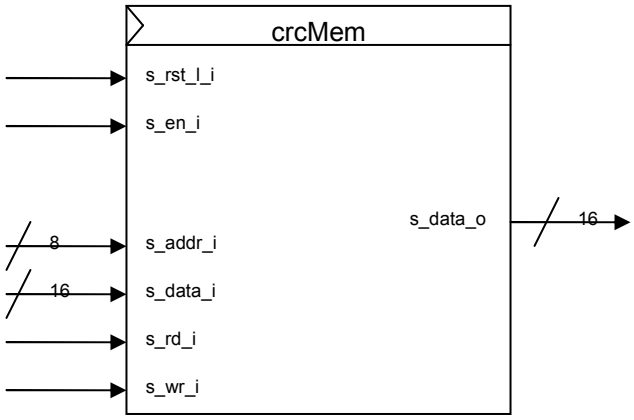

Figure 56 – crcMem module

2.2.2. Interface Description

The following table describes the clock(s) of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
t_clk_i	1	I	t_clk_i	-	-	Master Clock
s_rst_l_i	1	I	t_clk_i	L	-	Master Reset
s_en_i	1	I	t_clk_i	H	-	Master enable
s_addr_i	8	I	t_clk_i	-	-	Address input
s_data_i	16	I	t_clk_i	-	-	Input data bus
s_rd_i	1	I	t_clk_i	H	-	Read from memory to output data bus
s_wr_i	1	I	t_clk_i	H	-	Write from input data bus to memory
s_data_o	16	O	t_clk_i	-	16'hFFFF	Output data bus

Table 24 – crcMem Interface

2.2.3. Implementation

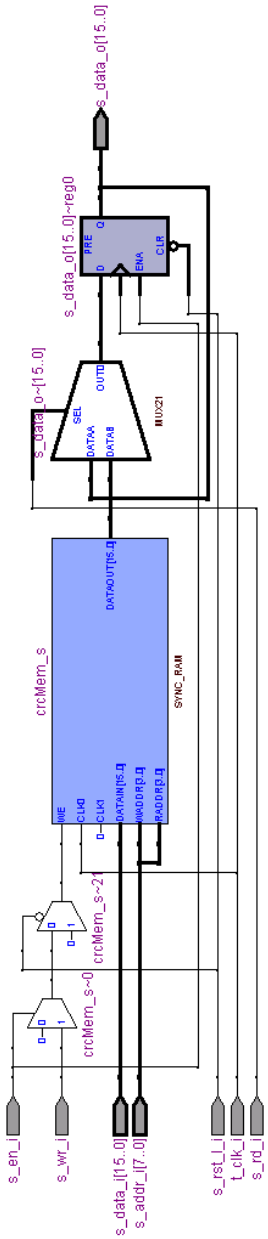

Figure 57 – crcMem RTL

2.3. Cyclic Redundancy Check Validation – crcChk IF (Level 1)

2.3.1. Overview

This block contains the logic necessary to compare a 16 bit CRC calculation. The stored CRCs, which are located in the CRC memory, are fed into crcChk logic and used for comparison against CRC values which are fed from another source, such as another TMU. This block merely acts as a value comparator and does not operate or execute any calculations over the CRC values. The following figure illustrates the available component interfaces.

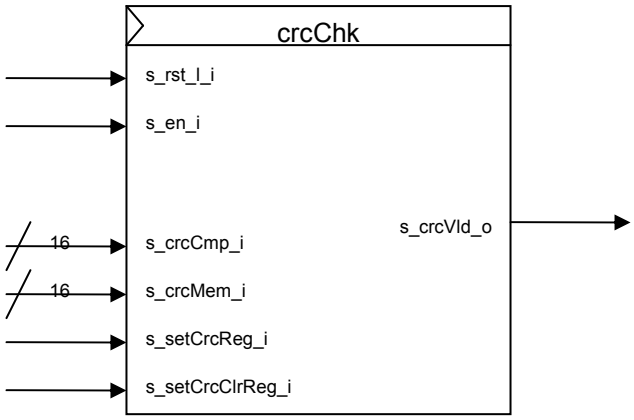

Figure 58 – crcChk module

2.3.2. Interface Description

The following table describes the signals of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
t_clk_i	1	I	t_clk_i	-	-	Master Clock
s_rst_l_i	1	I	t_clk_i	L	-	Master Reset
s_en_i	1	I	t_clk_i	H	-	Master enable
s_crcCmp_i	16	I	t_clk_i	-	-	External CRC
s_crcMem_i	16	I	t_clk_i	-	-	CRC from CRC memory
s_setCrcReg_i	1	I	t_clk_i	H	-	Internal buffer that stores the s_crcCmp_i value
s_setCrcClrReg_i	1	I	t_clk_i	H	-	Clear internal buffer
s_crcVld_o	1	O	t_clk_i	H	0	Valid CRC indicates if the result of the comparison is true (H) or false (L)

Table 25 – crcChk Interface

2.3.3. Implementation

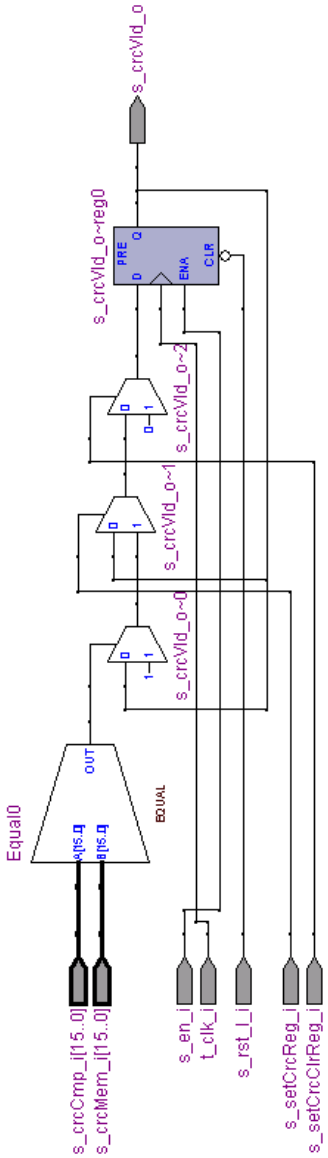

Figure 59 – crcChck RTL

2.4. Timer – timer IF (Level 1)

2.4.1. Overview

This block contains the logic necessary to execute a timer function similar to a watchdog timeout. Timer output is raised when a predefined value is reached (timeout). The timer output is asserted to low upon reset and until the predefined value is not reached. The predefined value is defined in the entity as a generic constant. The following figure illustrates the available component interfaces.

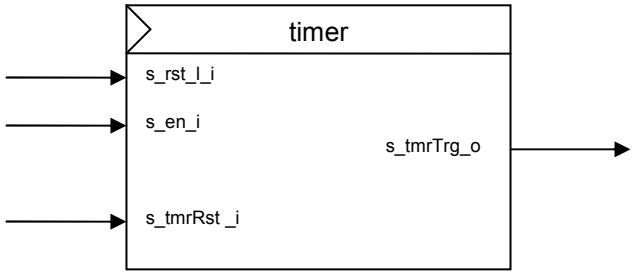

Figure 60 – timer module

2.4.2. Interface Description

The following table describes the signals of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
t_clk_i	1	I	t_clk_i	-	-	Master Clock
s_rst_l_i	1	I	t_clk_i	L	-	Master Reset
s_en_i	1	I	t_clk_i	H	-	Master enable
s_tmrRst_i	1	I	t_clk_i	H	-	Timer reset restart the internal counter used for the Timer
s_tmrTrg_o	1	O	t_clk_i	H	1'b0	Timer trigger is asserted upon timeout and will maintain this logic state until a reset is issued.

Table 26 – timer Interface

2.4.3. Implementation

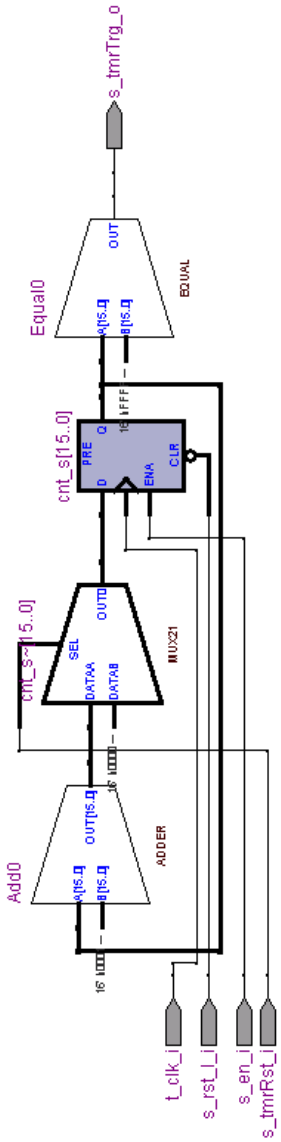

Figure 61 – timer RTL

2.5. Data and Address Coder/Decoder – datAddrDec IF (Level 1)

2.5.1. Overview

This module adapts the width of all data coming from the medium (i.e. from another TMU) or memory to the desired output (medium or memory) defined by the DT3C FSM. Besides the functionality of a cross connect between medium and memory, the datAddrDec also contains the necessary logic to calculate the addresses for the search algorithm and generate the correct addresses for memory access. The following figure illustrates the available component interfaces.

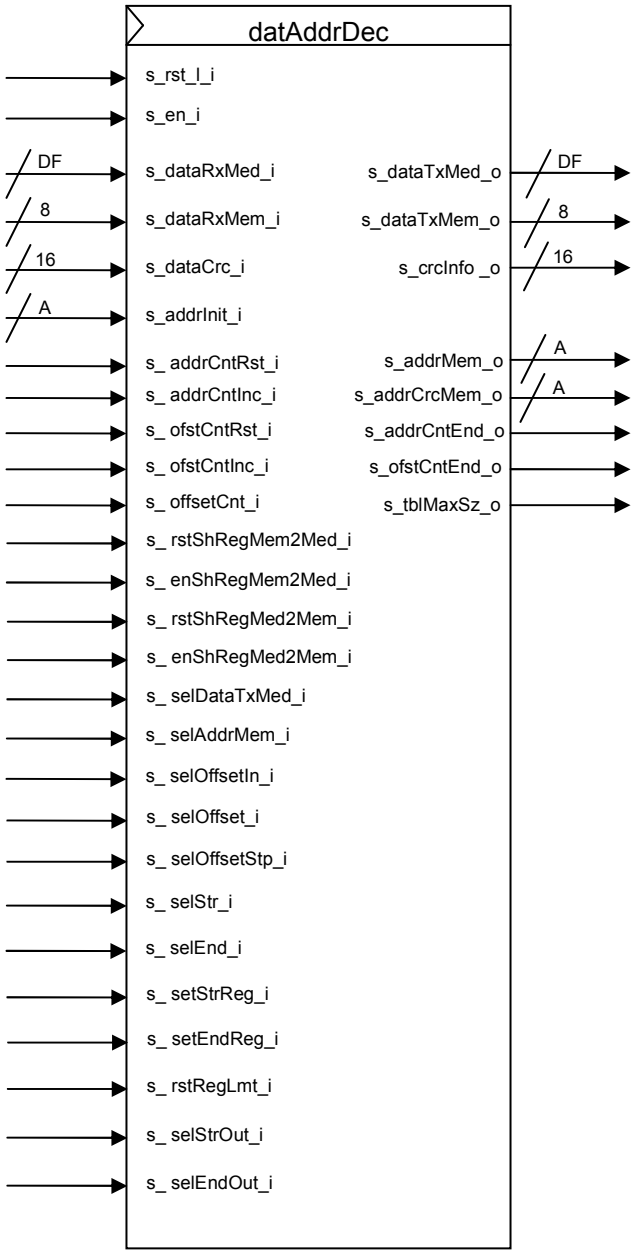

Figure 62 – datAddrDec RTL

2.5.2. Interface Description

Name	Width	Dir	Domain	Activity	Rst Value	Description
t_clk_i	1	I	t_clk_i	-	-	Master Clock.
s_rst_l_i	1	I	t_clk_i	L	-	Master Reset.
s_en_i	1	I	t_clk_i	H	-	Master enable.
s_dataRxMed_i	DF	I	t_clk_i	-	-	Data bus from Rx medium.
s_dataRxMem_i	8	I	t_clk_i	-	-	Data bus from Rx memory.
s_dataCrc_i	16	I	t_clk_i	-	-	Data from CRC bus.
s_addrInit_i	A	I	t_clk_i	-	-	Base address for memory access.
s_addrCntRst_i	1	I	t_clk_i	H	-	Address counter reset.
s_addrCntInc_i	1	I	t_clk_i	H	-	Address counter increment.
s_ofstCntRst_i	1	I	t_clk_i	H	-	Address offset counter reset.
s_ofstCntInc_i	1	I	t_clk_i	H	-	Address offset counter increment.
s_offsetCnt_i	Of	I	t_clk_i	-	-	Address offset counter.
s_rstShRegMem2Med_i	1	I	t_clk_i	H	-	Reset memory to medium.
s_enShRegMem2Med_i	1	I	t_clk_i	H	-	Enable memory to medium.
s_rstShRegMed2Mem_i	1	I	t_clk_i	H	-	Reset medium to memory.
s_enShRegMed2Mem_i	1	I	t_clk_i	H	-	Enable medium to memory.
s_setShRegMed2Mem_i	1	I	t_clk_i	H	-	Set medium to memory.
s_selDataTxMed_i	1	I	t_clk_i	H	-	Select transmit data from medium.
s_selAddrMem_i	1	I	t_clk_i	H	-	Select memory address.
s_selOffsetIn_i	2	I	t_clk_i	-	-	Select address offset input.
s_selOffset_i	1	I	t_clk_i	H	-	Select address offset.
s_selOffsetStp_i	1	I	t_clk_i	H	-	Select addressoffset end value.
s_selStr_i	2	I	t_clk_i	-	-	Select memory start address
s_selEnd_i	2	I	t_clk_i	-	-	Select memory end address
s_setStrReg_i	1	I	t_clk_i	H	-	Set start register.
s_setEndReg_i	1	I	t_clk_i	H	-	Set end register.
s_rstRegLmt_i	1	I	t_clk_i	H	-	Reset register limit value.
s_selStrOut_i	1	I	t_clk_i	H	-	Select start value output.
s_selEndOut_i	1	I	t_clk_i	H	-	Select end value output.
s_dataTxMed_o	DF	O	t_clk_i	-	(others => '0')	Medium Tx data.
s_addrMem_o	A	O	t_clk_i	-	(others => '0')	Address bus for memory.
s_addrCrcMem_o	A	O	t_clk_i	-	(others => '0')	Address bus for CRC memory.
s_dataTxMem_o	8	O	t_clk_i	-	(others => '0')	Memory Tx data.
s_crcInfo_o	16	O	t_clk_i	-	(others => '0')	CRC bus.
s_addrCntEnd_o	1	O	t_clk_i	H	0	Address count end value.
s_ofstCntEnd_o	1	O	t_clk_i	H	0	Address offset count end value.
s_tblMaxSz_o	Of	O	t_clk_i	-	0	Table maximum size.

Table 27 – datAddrDec Interface

2.5.3. Implementation

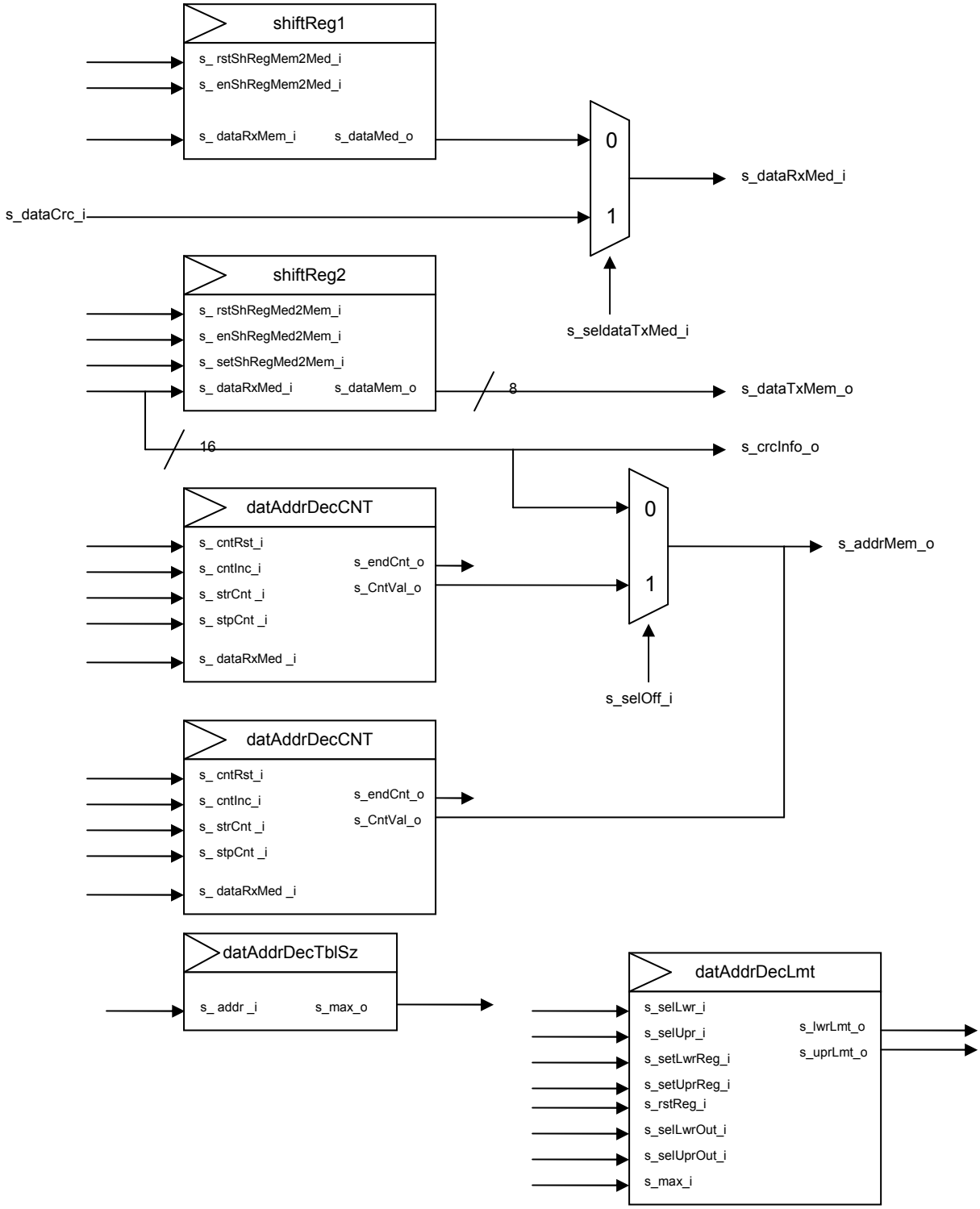

Figure 63 – datAddrDec internal components

2.5.4. datAddrDecCnt (Level 2)

This component is a counter and used for address and offset generation.

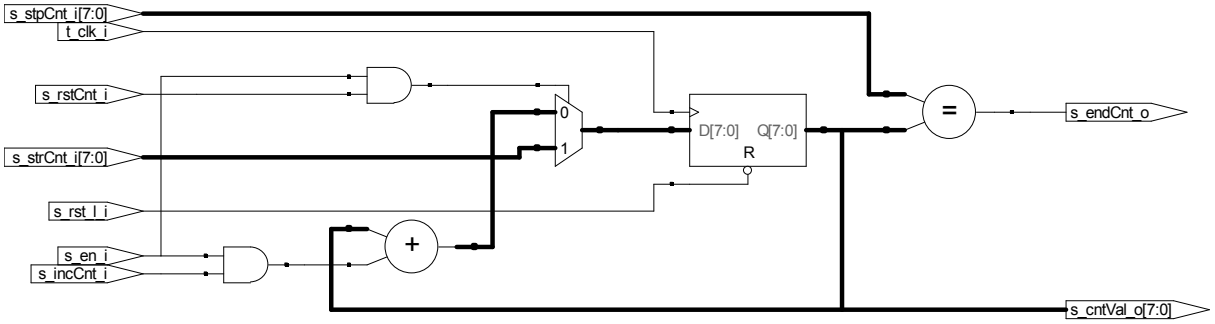

Figure 64 – datAddrDecCnt RTL

2.5.5. datAddrDecTblSz (Level 2)

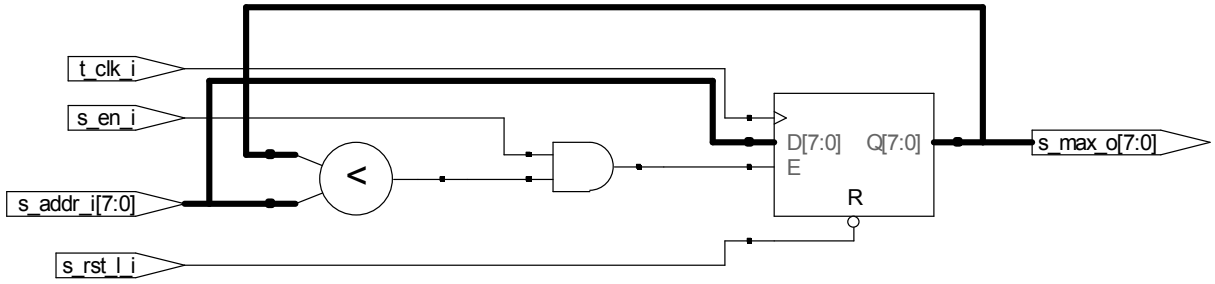

Figure 65 – datAddrDecTblSz RTL

2.5.6. datAddrDecLmt (Level 2)

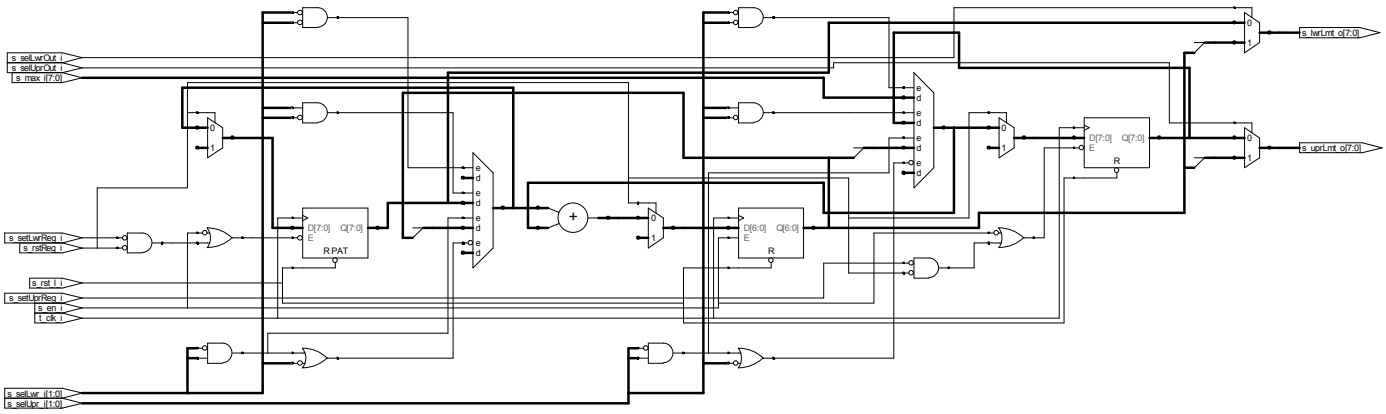

Figure 66 – datAddrDecLmt RTL

2.6. DT3C Finite State Machines – DT3CFSM IF (Level 1)

2.6.1. Overview

The DT3C FSM contains all finite state machines that allow the execution of the eight functionalities defined above and controls all the other blocks. The input control signals (commands) are connected to the main FSM which triggers all the secondary FSMs. All basic tasks such as retrieving information from memory or calculating the BoI CRCs are implemented in secondary FSMs.

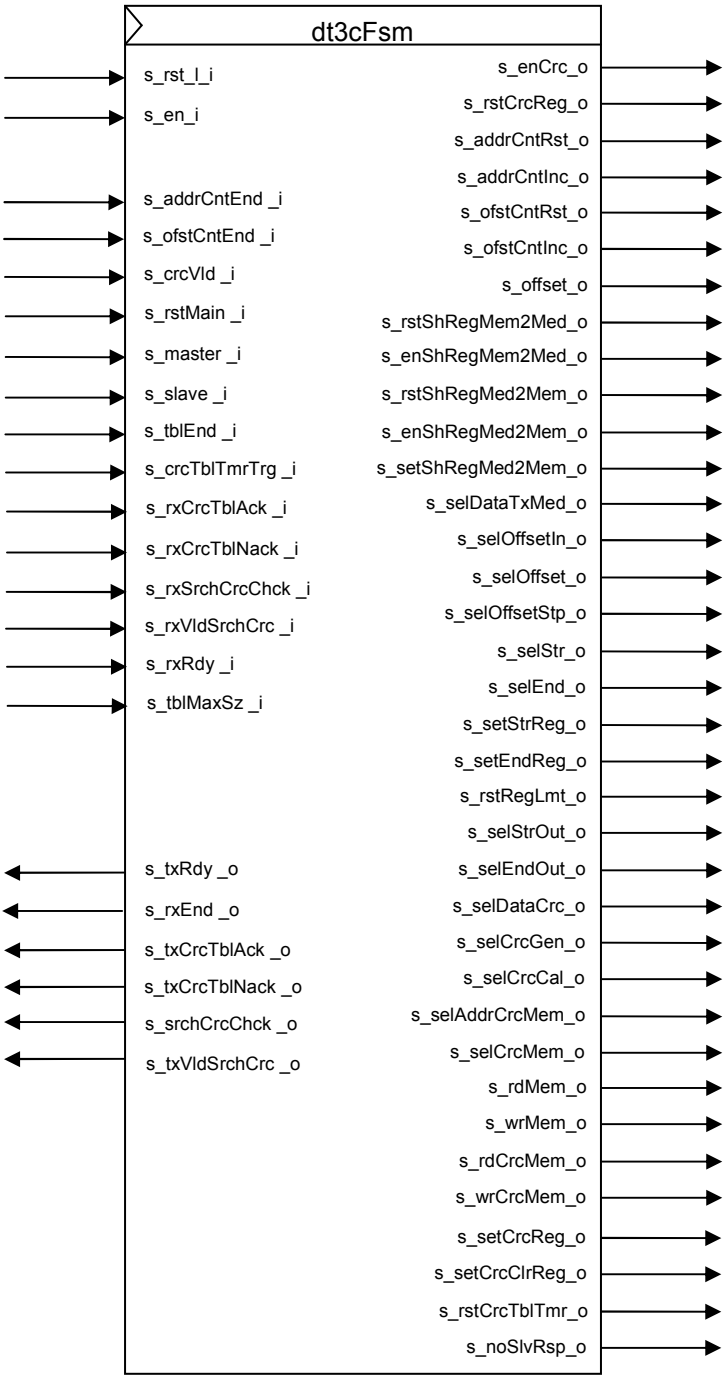

Figure 67 – dt3cFsm RTL

2.6.2. Interface Description

The following table describes the clock(s) of the present interface.

Name	Width	Dir	Domain	Activity	Rst Value	Description
t_clk_i	1	I	t_clk_i	-	-	Master Clock
s_rst_l_i	1	I	t_clk_i	L	-	Master Reset
s_en_i	1	I	t_clk_i	H	-	Master enable
s_addrCntEnd_i	1	I	t_clk_i	H	-	Address counter end indication
s_ofstCntEnd_i	1	I	t_clk_i	H	-	Offset Address counter end indication
s_crcVld_i	1	I	t_clk_i	H	-	CRC valid.
s_rstMain_i	1	I	t_clk_i	H	-	Reset main FSM.
s_master_i	1	I	t_clk_i	H	-	Master mode. Indicates that the DT3C is in master mode.
s_slave_i	1	I	t_clk_i	H	-	Slave mode. Indicates that the DT3C is in slave mode.
s_tblEnd_i	1	I	t_clk_i	H	-	Table end alarm.
s_crcTblTmrTrg_i	1	I	t_clk_i	H	-	CRC table timer trigger alarm.
s_rxCrcTblAck_i	1	I	t_clk_i	H	-	Receive CRC Table acknowledge.
s_rxCrcTblNack_i	1	I	t_clk_i	H	-	Receive CRC Table not acknowledge.
s_rxSrchCrcChck_i	1	I	t_clk_i	H	-	Receive Search CRC Check.
s_rxVldSrchCrc_i	1	I	t_clk_i	H	-	Receive Valid Search CRC.
s_rxRdy_i	1	I	t_clk_i	H	-	Receive ready alarm.
s_tblMaxSz_i	Of	I	t_clk_i	-	-	Table maximum size.
s_enCrc_o	1	O	t_clk_i	H	0	Enable CRC logic
s_rstCrcReg_o	1	O	t_clk_i	H	0	Reset CRC Register
s_addrCntRst_o	1	O	t_clk_i	H	0	Address counter reset
s_addrCntInc_o	1	O	t_clk_i	H	0	Address counter increment
s_ofstCntRst_o	1	O	t_clk_i	H	0	Offset address counter reset
s_ofstCntInc_o	1	O	t_clk_i	H	0	Offset address counter increment
s_offset_o	Of	O	t_clk_i	-	(others => ‘0’)	Offset address
s_rstShRegMem2Med_o	1	O	t_clk_i	H	0	Reset memory to medium
s_enShRegMem2Med_o	1	O	t_clk_i	H	0	Enable memory to medium
s_rstShRegMed2Mem_o	1	O	t_clk_i	H	0	Reset medium to memory
s_enShRegMed2Mem_o	1	O	t_clk_i	H	0	Enable medium to memory
s_setShRegMed2Mem_o	1	O	t_clk_i	H	0	Set medium to memory
s_selDataTxMed_o	1	O	t_clk_i	H	0	Select data from transmit medium
s_selOffsetIn_o	2	O	t_clk_i	-		Select offset address from input
s_selOffset_o	1	O	t_clk_i	H	0	Select offset address
s_selOffsetStp_o	1	O	t_clk_i	H	0	Select offset address end
s_selStr_o	2	O	t_clk_i	-	(others => ‘0’)	Select memory start address
s_selEnd_o	2	O	t_clk_i	-	(others => ‘0’)	Select memory end address
s_setStrReg_o	1	O	t_clk_i	H	0	Set start register. Store start value in register.

s_setEndReg_o	1	O	t_clk_i	H	0	Set end register. Store end value in register.
s_rstRegLmt_o	1	O	t_clk_i	H	0	Reset register limit value.
s_selStrOut_o	1	O	t_clk_i	H	0	Select start value output.
s_selEndOut_o	1	O	t_clk_i	H	0	Select end value output.
s_selDataCrc_o	1	O	t_clk_i	H	0	Select data from CRC path.
s_selCrcGen_o	2	O	t_clk_i	-	(others => '0')	Select CRC generator.
s_selCrcCal_o	1	O	t_clk_i	H	0	Select calculated CRC.
s_selAddrCrcMem_o	1	O	t_clk_i	H	0	Select address CRC for memory Access.
s_selCrcMem_o	1	O	t_clk_i	H	0	Select CRC memory.
s_rdMem_o	1	O	t_clk_i	H	0	Read memory strobe.
s_wrMem_o	1	O	t_clk_i	H	0	Write memory strobe.
s_rdCrcMem_o	1	O	t_clk_i	H	0	Read CRC memory strobe.
s_wrCrcMem_o	1	O	t_clk_i	H	0	Write CRC memory strobe.
s_setCrcReg_o	1	O	t_clk_i	H	0	Set CRC register.
s_setCrcClrReg_o	1	O	t_clk_i	H	0	Clear CRC register.
s_rstCrcTblTmr_o	1	O	t_clk_i	H	0	Reset CRC table timer.
s_noSlvRsp_o	1	O	t_clk_i	H	0	Raise slave response not present alarm.
s_txRdy_o	1	O	t_clk_i	H	0	Ready for transmission.
s_rxEnd_o	1	O	t_clk_i	H	0	End of reception alarm.
s_txCrcTblAck_o	1	O	t_clk_i	H	0	Transmit CRC table acknowledge.
s_txCrcTblNack_o	1	O	t_clk_i	H	0	Transmit CRC table not acknowledge.
s_srchCrcChck_o	1	O	t_clk_i	H	0	Initiate search CRC check FSM.
s_txVldSrchCrc_o	1	O	t_clk_i	H	0	Transmit Valid search check.

Table 28 – dt3cFsm Interface

2.6.3. Implementation

The following finite state machines are implemented in the DT3C:

- Main FSM - **DT3CFSMMain**
- Tx Table FSM - **DT3CFSMTxTbl**
- Tx BoI FSM - **DT3CFSMTxBoi**
- Rx BoI FSM - **DT3CFSMRxBoi**
- Tx CRC FSM - **DT3CFSMTxCrc**
- Rx CRC FSM - **DT3CFSMRxCrc**
- Calculate CRC Table FSM - **DT3CFSMCalCrcTbl**
- Get BoI Mismatch FSM - **DT3CFSMGetBoiMism**
- Tx Search CRC Check - **DT3CFSMTxSrchCrcChck**
- Rx Search CRC Check - **DT3CFSMRxSrchCrcChck**

2.6.3.1. DT3CFSMMain (Level 2)

This is the main FSM of the DT3C and it is responsible for setting the DT3C behavior in master or slave mode as described in chapter 5.4 - DT3C Behavioral Description. All the FSM present in the DT3C are dependent and controlled by this main FSM.

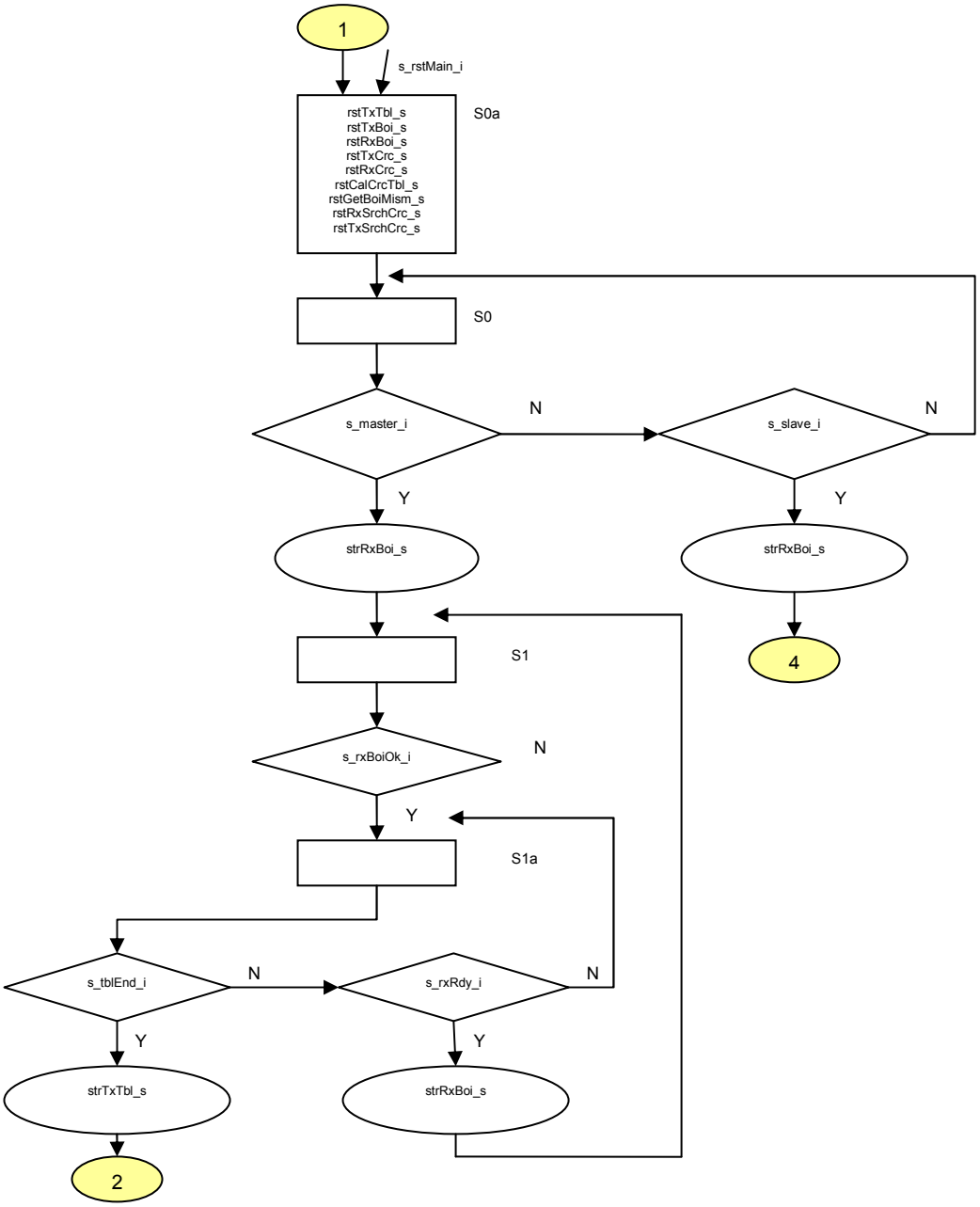

Figure 68 – DT3CFSMMain 1/3

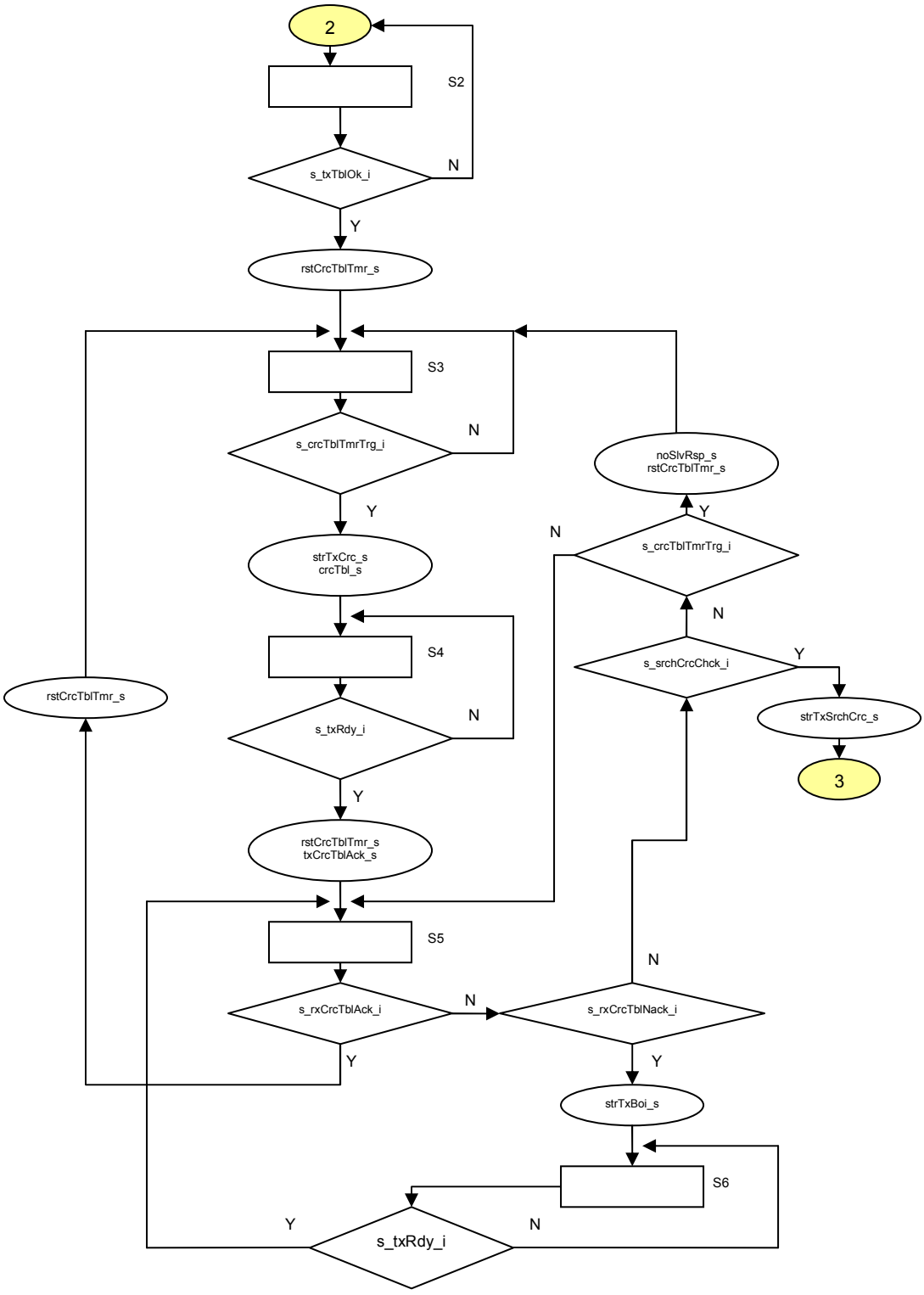

Figure 69 – DT3CFSMMain 2/3

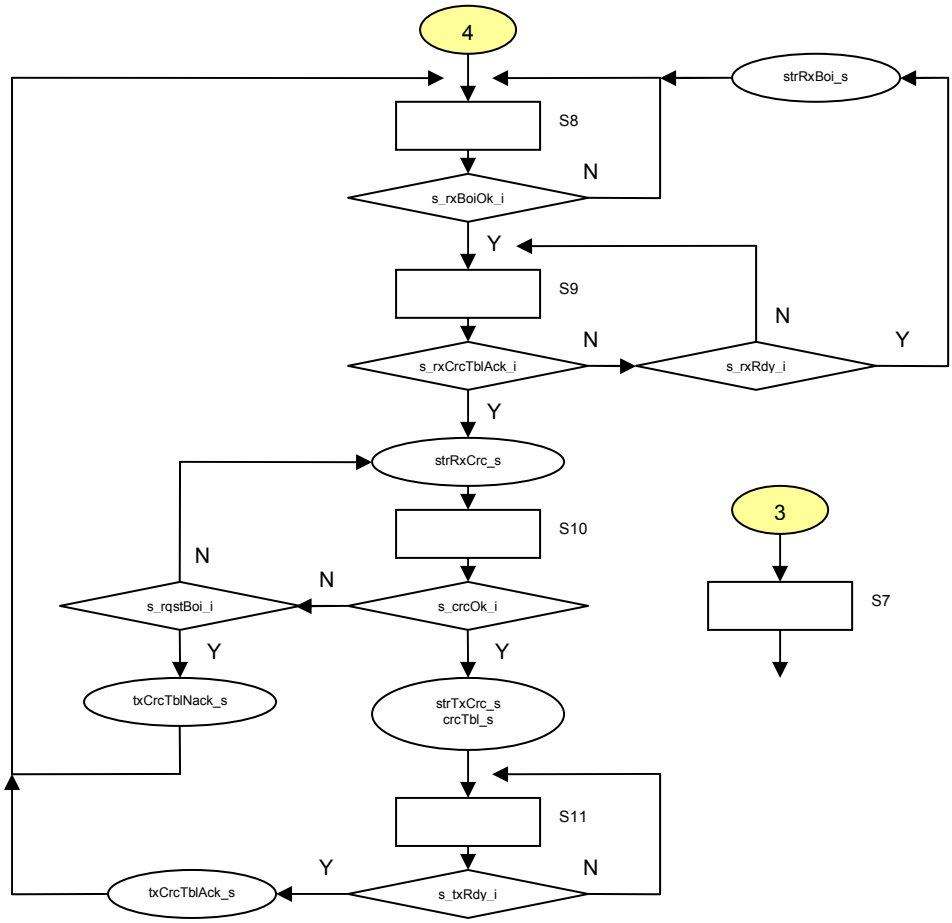

Figure 70 – DT3CFSMMain 3/3

2.6.3.2. DT3CFSMTxTbl (Level 2)

The DT3CFSMTxTbl is responsible for dispatching the table to the DT3C slave. During this process, the data integrity in the table is checked for any mismatch as described in chapter 5.4 - DT3C Behavioral Description. This FSM will only be executed in master mode and makes use of the DT3CFSMTxBoi finite state machine.

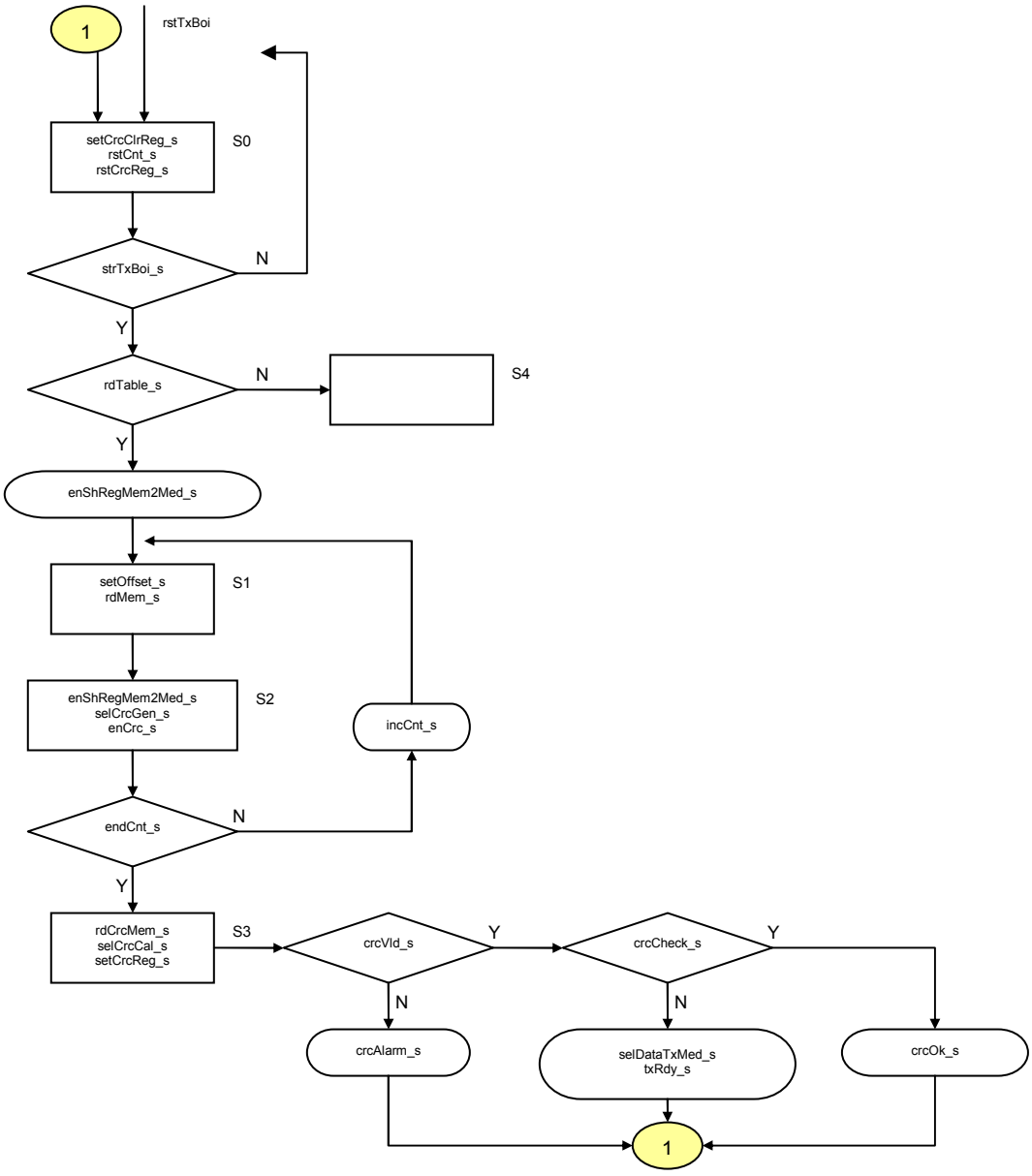

Figure 71 – DT3CFSMTxTbl

2.6.3.3. DT3CFSMTxBoi (Level 2)

The DT3CFSMTxBoi is responsible for reading each Boi from the table memory and placing the data on the external data bus (medium bus).

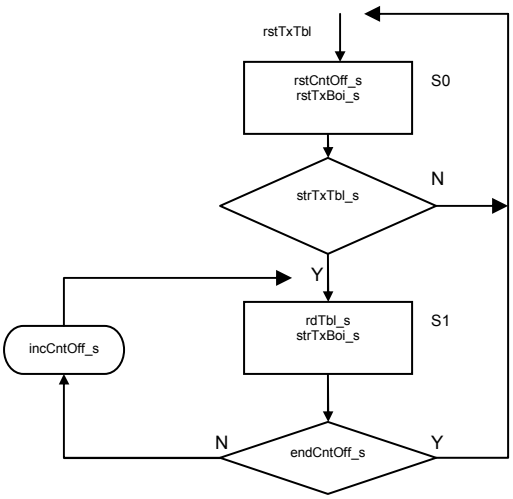

Figure 72 – DT3CFSMTxBoi

2.6.3.4. DT3CFSMRxBoi (Level 2)

The DT3CFSMRxBoi is responsible for reading data from the external bus (medium bus), feeding this data into the crcGen, storing the calculated CRC into the CRC memory and storing the Boi into the table memory.

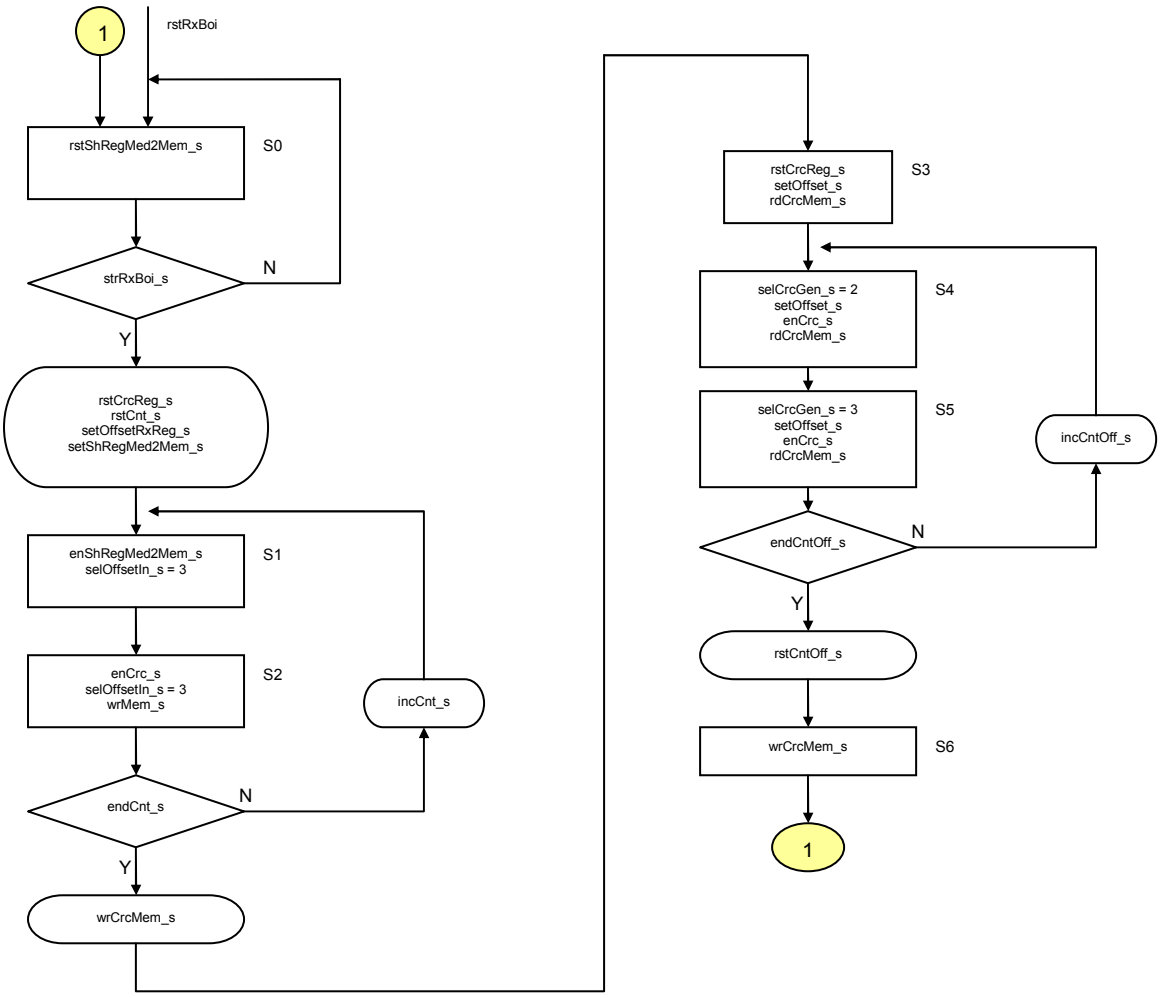

Figure 73 – DT3CFSMRxBoi

2.6.3.5. DT3CFSMTxCrc (Level 2)

The DT3CFSMTxCrc is responsible for reading data from the CRC memory and placing the CRC data on the external data bus (medium bus).

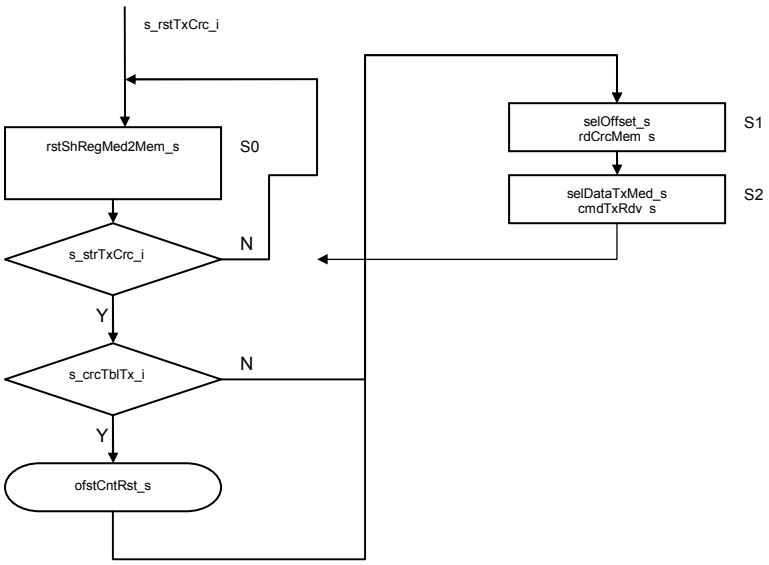

Figure 74 – DT3CFSMTxCrc

2.6.3.6. DT3CFSMRxCrc (Level 2)

The DT3CFSMRxCrc is responsible for reading data from the CRC memory and calculating the CRC from the corresponding Boi. Both CRC values are fed into the crcCmp logic. This FSM will initiate DT3CFSMGetBoiMism upon a CRC mismatch.

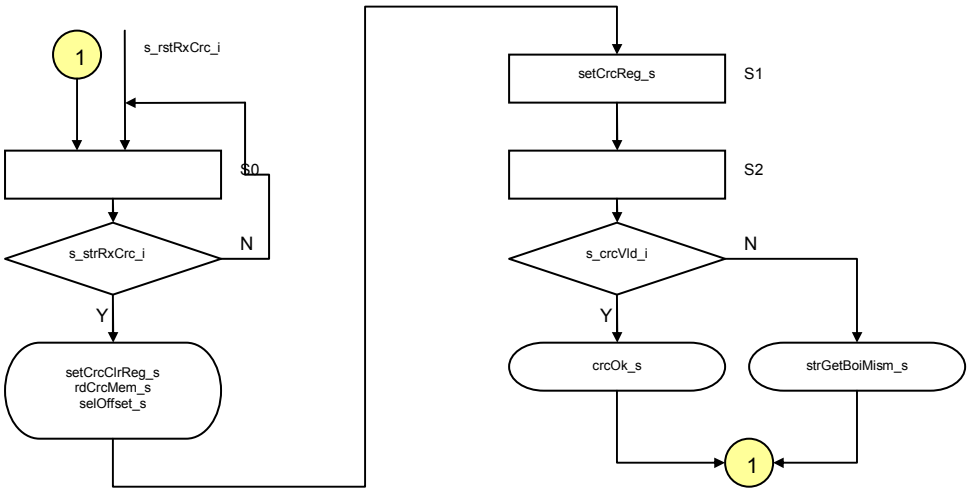

Figure 75 – DT3CFSMRxCrc

2.6.3.7. DT3CFSMCalCrcTbl (Level 2)

The DT3CFSMCrcTbl is responsible for reading data from the CRC memory and calculating a CRC for a predetermined range established by DT3CFSMTxSrchCrcChck or DT3CFSMRxSrchCrcChck.

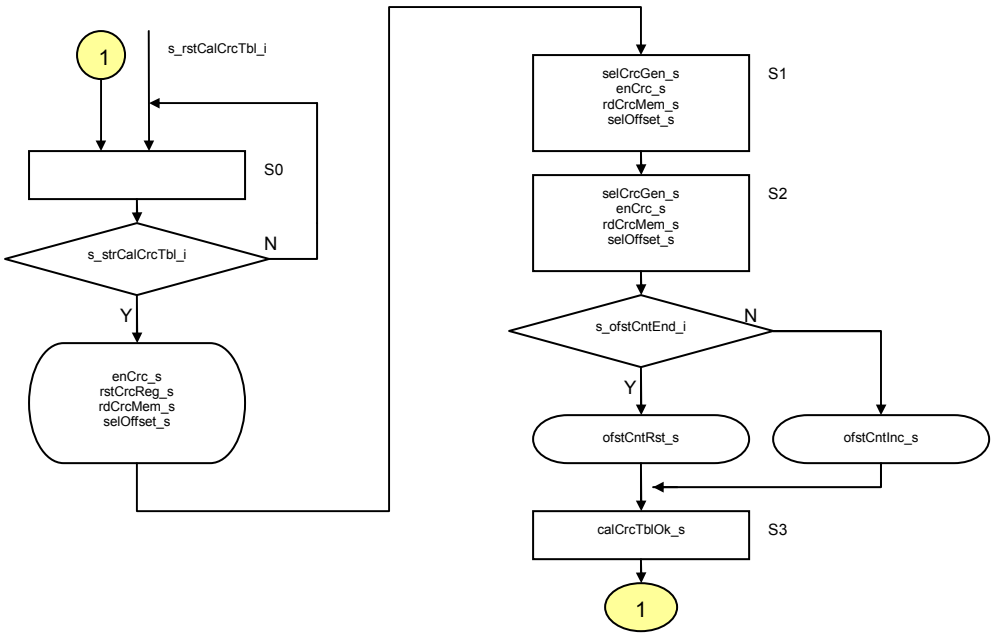

Figure 76 – DT3CFSMCalCrcTbl

2.6.3.8. DT3CFSMGetBoiMism (Level 2)

The DT3CFSMGetBoiMism is activated by the DT3CFSMRxCrc finite state machine. This FSM is responsible for determining inconsistencies between the data in the table memory and its CRC calculation, which resides in the CRC memory, thus executing the functionality of the ICV task described in chapter 5.4 - DT3C Behavioral Description.

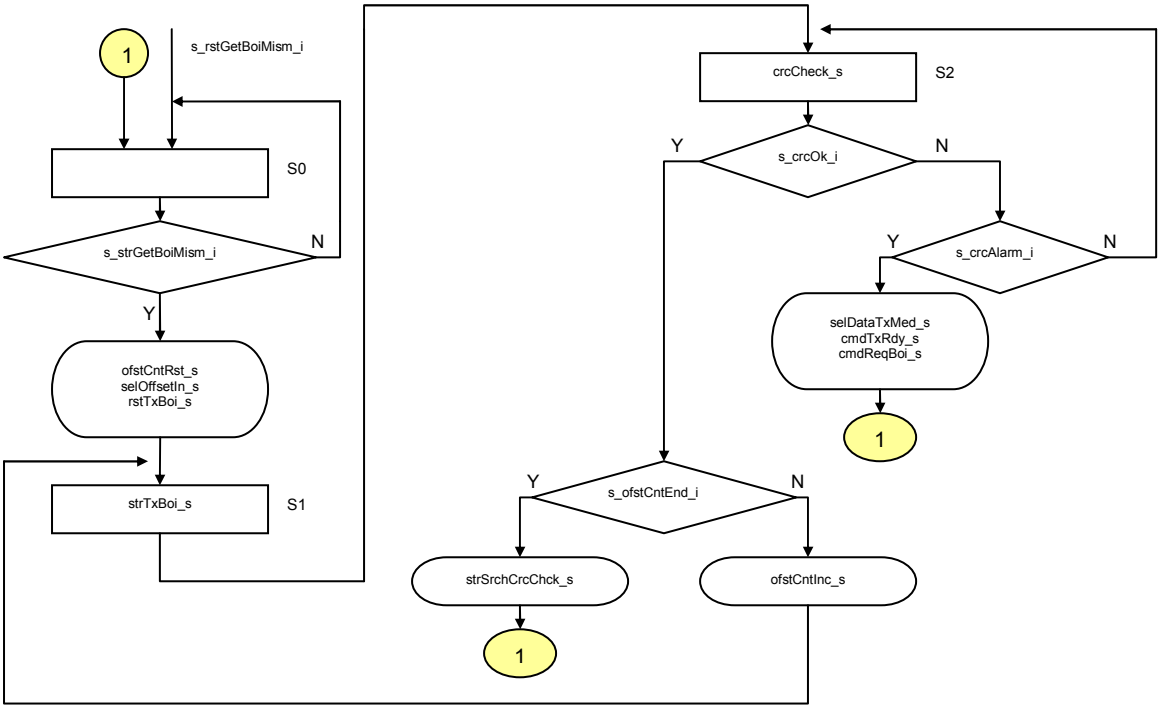

Figure 77 – DT3CFSMGetBoiMism

2.6.3.9. DT3CFSMTxSrchCrcChck (Level 2)

The DT3CFSMTxSrchCrcChck executes the functionality of the CMC task described in chapter 5.4 - DT3C Behavioral Description in the DT3C slave. This finite state machine will not be used (maintain state s0) when the DT3C is configured in master mode.

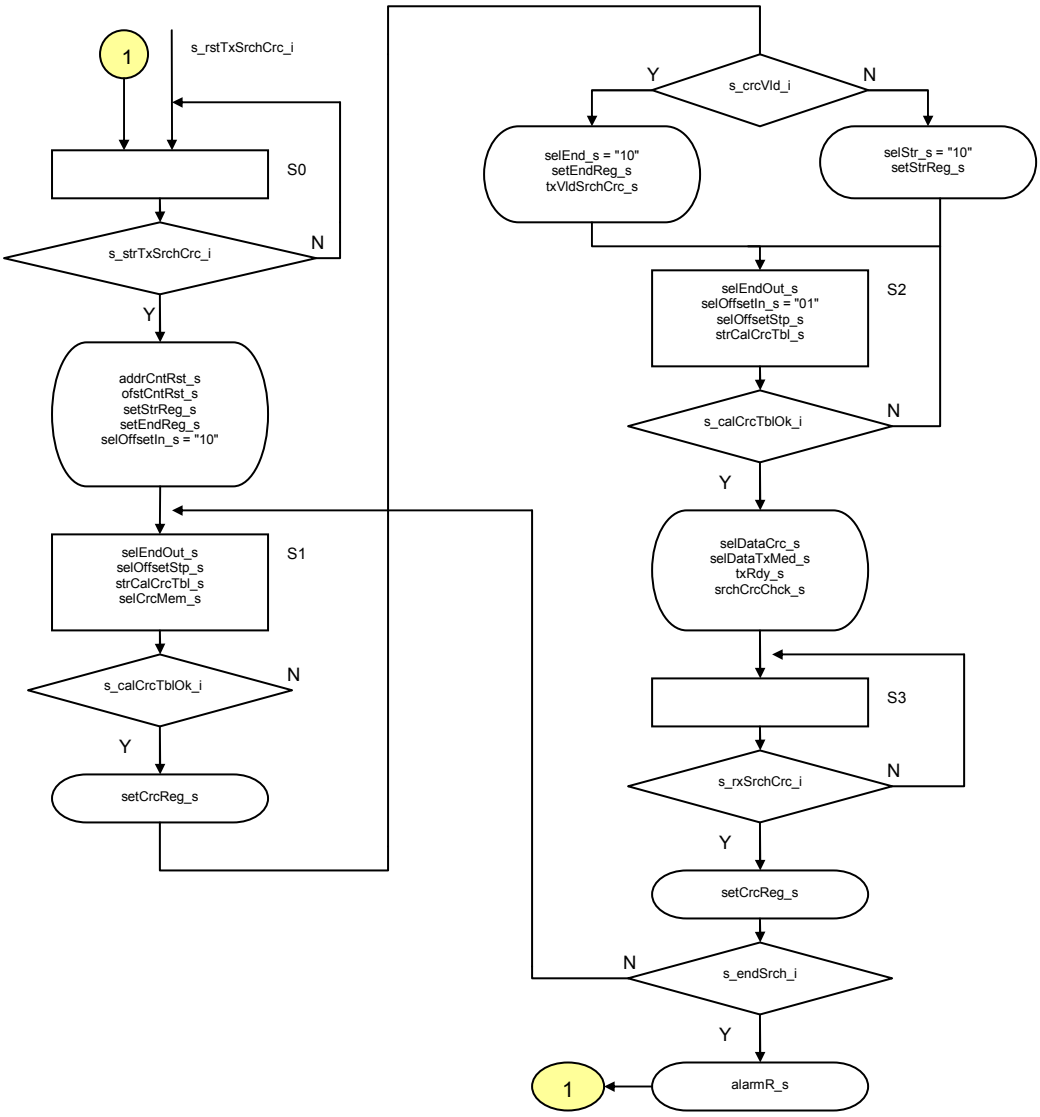

Figure 78 – DT3CFSMTxSrchCrcChck

2.6.3.10. DT3CFSMRxSrchCrcChck (Level 2)

The DT3CFSMRxSrchCrcChck executes the functionality of the CMC task described in chapter 5.4 - DT3C Behavioral Description in the DT3C master. This finite state machine will not be used (maintain state s0) when the DT3C is configured in slave mode.

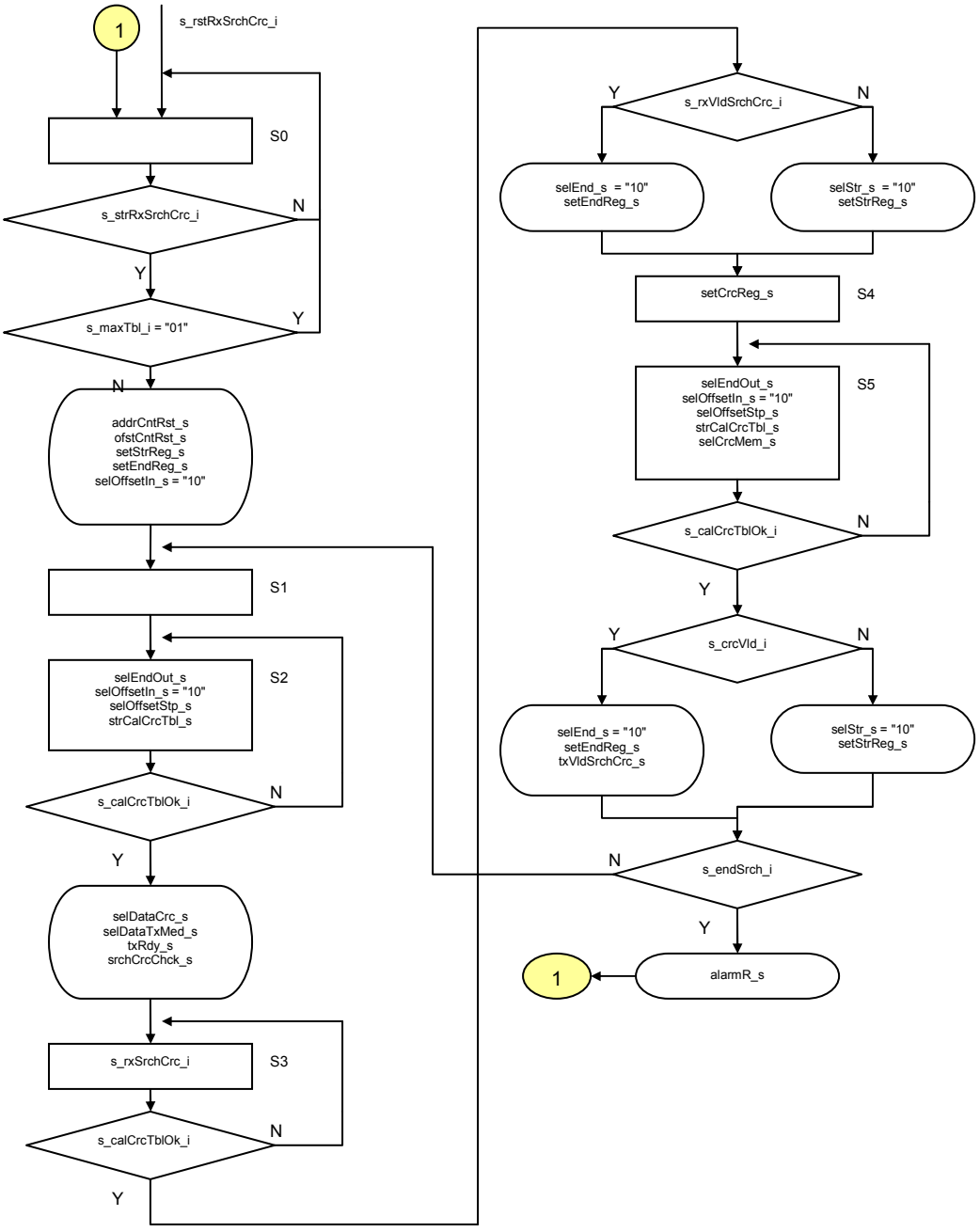

Figure 79 – DT3CFSMRxSrchCrcChck

A4. DT3C System Testbench

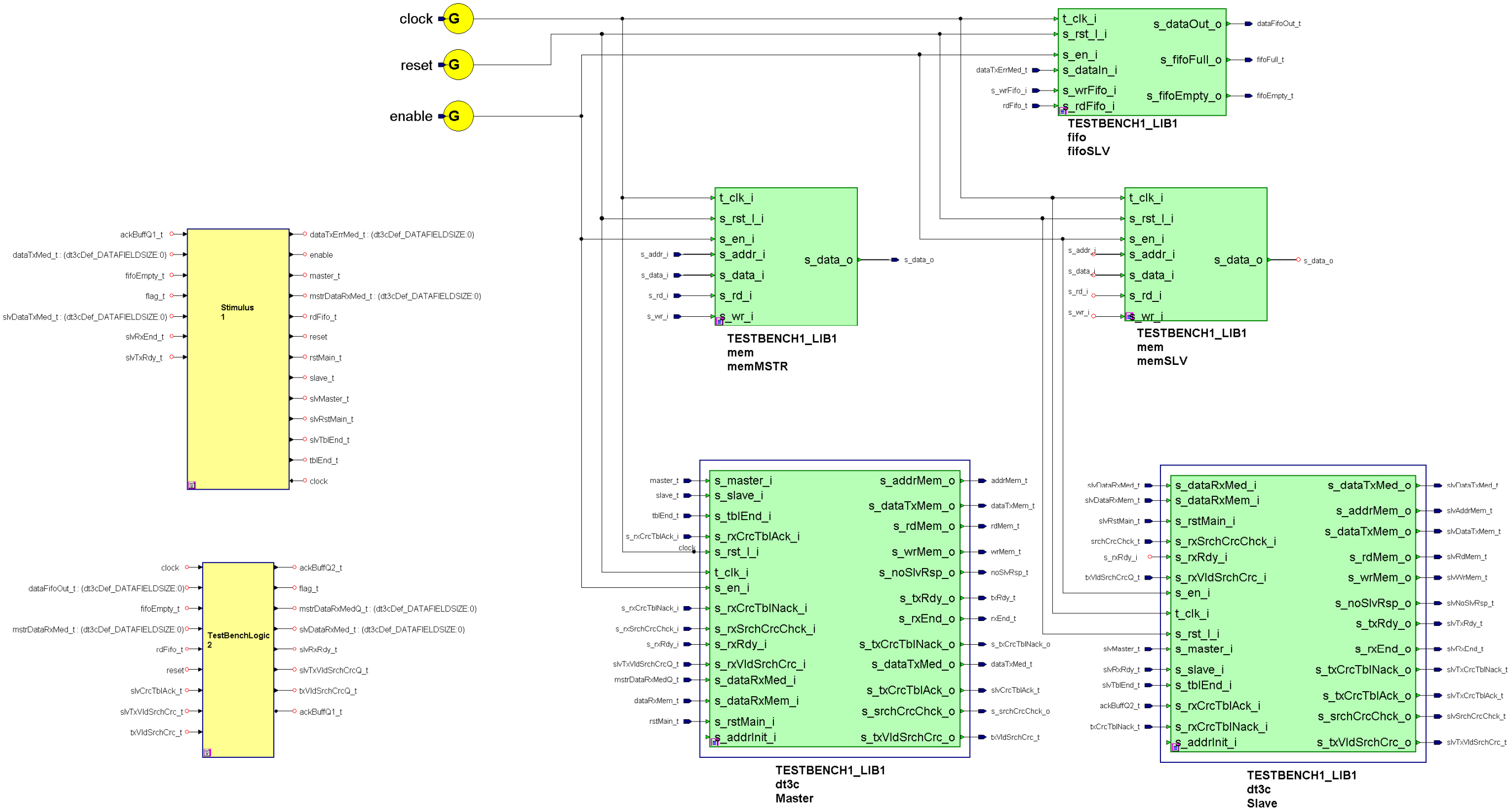

Figure 80 – DT3C Module Simulation

A5. CMC Algorithm

```
%------------------------------------------------------------------------
%CMC algorithm emulator
%
%------------------------------------------------------------------------
function out = cmc_algorithm()

%clear all
%clc

%master and slave memory vectors
[cmc_crcmemMstr, cmc_crcmemSlv] = build;

x_mst = cmc_crcmemMstr;
x_slv = cmc_crcmemSlv;


%init variables
x_slv_chck = 0;
x_slv_low  = 0;
x_slv_hgh  = 0;
slv_lstr   = 0;
slv_hstr   = 0;
x_mst_chck = 0;
x_mst_low  = 0;
x_mst_hgh  = 0;
end_cycle  = 0;

slv_msg_cnt = 0;
mst_msg_cnt = 0;
total_msg_cnt = 0;
flag_msg = 0;

%set to default
x_slv_len = length(x_slv)-1;
slv_lstr  = 1;
slv_lstp  = floor(x_slv_len/2);
slv_hstr  = floor(x_slv_len/2) + 1;
slv_hstp  = x_slv_len;

x_mst_len = length(x_mst)-1;
mst_lstr  = 1;
mst_lstp  = floor(x_mst_len/2);
mst_hstr  = floor(x_mst_len/2) + 1;
mst_hstp  = x_mst_len;

while end_cycle == 0

    %msg counter
    if flag_msg == 0
        %slave sent msg
        flag_msg = 1;
        slv_msg_cnt = slv_msg_cnt + 1;
    else
        %slave sent msg
        flag_msg = 0;
        mst_msg_cnt = mst_msg_cnt + 1;
    end

    %slave operations - calculate CRC over the range
```

```
    x_slv_low   = x_slv(slv_lstr:slv_lstp,:);
    x_slv_hgh   = x_slv(slv_hstr:slv_hstp,:);
    x_slv_chck  = crc_range(x_slv_low);
    x_slv_chckh = crc_range(x_slv_hgh);

    %master operations - calculate CRC over the range
    x_mst_low   = x_mst(mst_lstr:mst_lstp,:);
    x_mst_hgh   = x_mst(mst_hstr:mst_hstp,:);
    x_mst_chck  = crc_range(x_mst_low);
    x_mst_chckh = crc_range(x_mst_hgh);

    if x_slv_chck == x_mst_chck
            % lower bank is equal : than the error is in the upper bank
            if slv_hstr ~= slv_hstp
                end_cycle = 0;
                upper = 1;
                slv_lstr  = slv_hstr;
                slv_lstp  = floor((slv_hstr + slv_hstp)/2);
                slv_hstr  = floor((slv_hstr + slv_hstp)/2) + 1;
                slv_hstp  = slv_hstp;

                mst_lstr  = mst_hstr;
                mst_lstp  = floor((mst_hstr + mst_hstp)/2);
                mst_hstr  = floor((mst_hstr + mst_hstp)/2) + 1;
                mst_hstp  = mst_hstp;
            else
                % leaf has been reached
                end_cycle = 1;
                display('Found incorrect CRC BOI in Slave:')
                crc_Boi=x_slv(slv_hstr,:)
                display('Located in Offset address:')
                slv_hstr
            end
    else
            % lower bank is NOT equal : error is in this bank
            if slv_lstr ~= slv_lstp
                end_cycle = 0;
                lower = 1;
                slv_hstr = floor((slv_lstr + slv_lstp)/2) + 1;
                slv_hstp = slv_lstp;
                slv_lstr = slv_lstr;
                slv_lstp = floor((slv_lstr + slv_lstp)/2);

                mst_hstr = floor((mst_lstr + mst_lstp)/2) + 1;
                mst_hstp = mst_lstp;
                mst_lstr = mst_lstr;
                mst_lstp = floor((mst_lstr + mst_lstp)/2);
            else
                % leaf has been reached
                end_cycle = 1;
                display('Found incorrect CRC BOI in Slave:')
                crc_Boi=x_slv(slv_lstr,:)
                display('Located in Offset address:')
                slv_lstr
            end
    end
end



if x_slv_low ~= x_mst_low
    mismatch = x_slv_low;
    location = slv_lstr;
elseif x_slv_hgh ~= x_mst_hgh
```

```
    mismatch = x_slv_hgh;
    location = slv_hstr;
else
    mismatch = -1;
    location = -1;
end

if flag_msg == 1
    % last msg sent was slave
%    mst_msg_cnt = mst_msg_cnt + 1;
else
    slv_msg_cnt = slv_msg_cnt + 1;
%     mst_msg_cnt = mst_msg_cnt + 1;
end
slv_msg_cnt;
mst_msg_cnt;
flag_msg;
total_msg_cnt = slv_msg_cnt + mst_msg_cnt;
display('Total number of exchanged packets:')
out = total_msg_cnt;




%-------------------------------------------------------------------------
%build memories and store dummy data
%
%-------------------------------------------------------------------------
function [crcmemMstr, crcmemSlv] = build()

[tblmemMstr, tblmemSlv] = b_tmem();
crcmemMstr = b_crcmem(tblmemMstr);
crcmemSlv = b_crcmem(tblmemSlv);




%-------------------------------------------------------------------------
%build table memories
%
%-------------------------------------------------------------------------
function [my_tblmemMstr, my_tblmemSlv] = b_tmem()

my_tblmemMstr=['0011'; '0022'; '0033'; '0044'; '0055'; '0066'; '0077'; '0088';
'0099'; '00aa'; '00bb'; '00cc'; '00dd'; '00ee'; '00ff';];
my_tblmemSlv=['1011'; '0022'; '0033'; '0044'; '0055'; '0066'; '0077'; '0088';
'0099'; '00aa'; '00bb'; '00cc'; '00dd'; '00ee'; '00ff';];




%-------------------------------------------------------------------------
%build crc memories
%
```

```
%-------------------------------------------------------------------------
function my_crcmem = b_crcmem(tablemem)

my_crcmem=['0000'; '0000'; '0000'; '0000'; '0000'; '0000'; '0000'; '0000';
'0000'; '0000'; '0000'; '0000'; '0000'; '0000'; '0000'; '0000';];
for i=1:length(tablemem)
    my_crcmem(i,:)=crc_range(tablemem(i,:)); %
end
my_crcmem(16,:)=crc_range(my_crcmem(1:15,:));



%-------------------------------------------------------------------------
%crc calculator - 16 bit CRC-CCITT (XModem) x^16 + x^12 + x^5 + 1
%
%-------------------------------------------------------------------------
function dataout = crc_range(hex_vector)

r=zeros(1,16);      % Remainder register initialization

[my_size,nop]=size(hex_vector);
for cnt=1:my_size

    hexvalue = hex_vector(cnt,:);
    my_len = length(hexvalue)*4;
    out1 = hex2dec(hexvalue);
    out2 = dec2binvec(out1,my_len);
    msg = out2(my_len:-1:1);

    for c3=1:length(msg)
        s1=bitxor(msg(c3),r(1));    % XOR between r0 and the message bit
        s2=bitxor(s1,r(12));        % XOR r11
        s3=bitxor(s1,r(5));         % XOR r4
        r=[r(2:16) s1];             % Left shift of r, and r15 update
        r(11)=s2;                   % r10 update
        r(4)=s3;                    % r3 update
    end
end

out3=r(16:-1:1);
out4=binvec2dec(out3);
dataout=dec2hex(out4,4);
```

Abbreviations

ABS	Anti-locking Brake System
Ack	Acknowledge
ARQ	Automatic Repeat Request
BFDunit	Bus Fault Detector unit – TMU internal submodule
BoI	Block of Information
CAN	Controller Area Network
CANBusIF	CAN Bus Interface
CMC	CRC Memory Comparison
CRC	Cyclic Redundancy Check
crcChk	CRC check
crcGen	CRC Generator
crcMem	CRC Memory
Ctrl	Control
datAddDec	Data and Address Decoder
DT3C	Distributable Table Content Consistency Checker
DTM	Dynamic Topology Management
EMI	Electromagnetic Emission
ESP	Electronic Stability Program
FDI	Fault Diagnose and Identification
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
FTT	Flexible Time Triggered
ICV	Internal CRC Verification
ID	Identification
IFM	Interface Manager
LUT	Look-Up Table
Mem	Memory
NSU	Network Switch Unit
QoS	Quality of Service
RAM	Random-access memory
RecMan	Reconfiguration Manager
RedMan	Redundancy Manager

Rd	Read
Rx	Receive
Tx	Transmit
TDR	Time Domain Reflectometry
TM	Trigger Message
TMU	Topology Manager Unit
TMUCAM	TMU Configurator and Monitor
UART	Universal Asynchronous Receiver-Transmitter
uC	Microcontroller
VHDL	VHSIC hardware description language
VHSIC	Very-High-Speed Integrated Circuits
Wr	Write

Definitions

The following definitions are addressed throughout the text:

Error - Cause of a fault.

Failure - Internal system state that does not correspond to specification, not visible at interfaces.

Fault - Deviation of system from specification at interfaces.

Dependability - ability to avoid failures that are more frequent or more severe, and outage durations that are longer, than is acceptable to the user/system.

Reliability - Continuity of correct service.

Availability - Readiness for correct service.

Safety - Absence of catastrophic failures.

Confidentiality - No unauthorized disclosure.

Integrity - No improper states or state changes.

Real-time - task that must be completed within an established time interval

References

[1] **ISO. Road Vehicles, Interchange of digital information – Controller Area Network (CAN) for high-speed communication**
International Standard ISO/IEC 11898
November, 1993

[2] **CAN Specification**
BOSCH
Version 2.0
1991

[3] **Dynamic Topology Management in CAN**
J. A. Fonseca, J. Ferreira, and V. F. Silva
Emerging Technologies and Factory Automation
September, 2006

[4] **Fundamental Concepts of Dependability**
A. Avizienis, J.C. Laprie, and B. Randell
Critical Systems Conference – Birmingham
October, 2001

[5] **Resilience, Reliability and Redundancy**
G. Marshall, and D. Chapman
Copper Development Association
May, 2002

[6] **An Analysable Bus-Guardian for Event-Triggered Communication**
I. Broster, and A. Burns
24th Real-Time Systems Symposium
December, 2003

[7] **A Columbus' Egg Idea for CAN Media Redundancy**
J. Rufino, P. Veríssimo, and G. Arroz
29th International Symposium on Fault-Tolerant Computing
June, 1999

[8] **RedCAN: Simulations of Two Fault Recovery Algorithms for CAN**
H. Sivencrona, O. Olsson, R. Johansson, J. Torin
10th IEEE Pacific Rim International Symposium on Dependable Computing
2004

[9] **CANcentrate: An Active Star Topology for CAN Networks**
M. Barranco, G. Rodriguez-Navas, J. Proença, and L. Almeida
Proceedings of the 5th IEEE International Workshop on Factory Communication Systems (WFCS)
2004

[10] **ReCANcentrate: A replicated star topology for CAN networks**
M. Barranco, J. Proença, and L. Almeida
Proceedings of the 2005 IEEE Conference on Emerging Technologies and Factory Automation
2005

[11] **A new CAN-like field network based on a star topology**
G. Cena, L. Duarte, and A. Valenzano
Comput. Stand. Interfaces
2001

[12] ***Optical Layer for CAN***
M. Rucks
1st International CAN Conference
1994

[13] **A new CAN-like field network based on a star topology**
G. Cena, L. Duarte, and A. Valenzano
Comput. Stand. Interfaces
2001

[14] **Overcoming Babbling-Idiot Failures in FlexCAN Architecture: A simple Bus-Guardian**
G. Buja, A. Zucocollo, J. Pimentel
Emerging Technologies and Factory Automation
September, 2005

[15] ***Using FTT-CAN to Combine Redundancy with Increased Bandwidth***
V. Silva and J. Fonseca
In Proceedings of the 2006 IEEE International Workshop on Factory Communication Systems, pages 54–62,
June, 2006

[16] ***Using* FTT-CAN to the Flexible Control of Bus Redundancy and Bandwidth Usage**
V. Silva, J. Fonseca, and J. Ferreira
In Proceedings of the 11th International CAN Conference iCC 2006, pages 5.9 – 5.15, Sweden,
September, 2006

[17] **Fault Containment and Error Detection in TTP/C and FlexRay**
H. Kopetz
Research Report, version 1.5
August 2002

[18] **Fault Detection in Fieldbuses with Time Domain Reflectometry**
M. Hartebrodt, and K. Kabitzsch
7th Africon Conference in Africa
2004

[19] **Neuro-Fuzzy Fault Detection Approach using a Profibus Network**
J. Calado, M. Kowal, M. Mendes, J. Korbicz, and J. Costa
Proceedings of the 10th Mediterranean Conference on Control and Automation
July, 2002

[20] **Fault Containment and Error Detection in TTP/C and FlexRay**
H. Kopetz
Technical University of Vienna, Research Report
August, 2002

[21] **Digital Design Guidelines**
Roland Holler
Arbeitsgruppe CAD, TU WIEN
2002

[22] **Error Control Coding: Fundamentals and Applications**
S. Lin, and D. J. Costello
Prentice-Hall
1983

[23] **Understanding Replication in Databases and Distributed Systems**
M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso
Proceedings of 20th International Conference on Distributed Computing Systems
2000

[24] **The FTT-CAN Protocol: Why and How**
L. Almeida, P. Pedreiras, and J. A. Fonseca
IEEE Transactions on Industrial Electronics
2002

[25] ***Achieving Fault Tolerance in FTT-CAN***
J. Pedreiras, P. Almeida, and L. Fonseca
4th IEEE International Workshop on Factory Communication Systems
2002

[26] **Automatic repeat request pages 181-198, Communications Handbook**
David Haccoun and Samuel Pierre
CRC Press / IEEE Press, Boca Raton, Florida
1996

[27] **Intermediate Checksums for Improving Goodput over Error-Prone Links**
Andreas Willig
In Proc. IEEE Vehicular Technology Conference (VTC)
2004

[28] **Error control schemes for networks: An overview**
Hang Liu, Hairuo Ma, Magda El Zarki, and Sanjay Gupta
MONET – Mobile Networks and Applications, 2(2):167–182
1997

[29] **Intermediate Checksum Schemes in the Presence of Deadlines**
Andreas Willig
Technical University Berlin, Telecommunication Networks Group
2008

[30] **Multi bit errors Vulnerabilities in the Controller Area Network Protocol**
Eushiuan Tran
MsC Dissertation, Carnegie Mellon University
1999

[31] **Error Control Systems for Digital Communication and Storage**
S. B. Wicker
Prentice-Hall, Inc.
1995